HumBird: Rotating Leader BFT made Simple and Fast

Chenxu Wang
Shandong University
Qingdao, China
cxwang16117@gmail.com

Sisi Duan*
Tsinghua University
Beijing, China
duansisi@tsinghua.edu.cn

Minghui Xu*
Shandong University
Qingdao, China
mhxu@sdu.edu.cn

Huizhong Li
ICT/CAS and UCAS
Beijing, China
WeBank Blockchain Team
Shenzhen, China
lihuizhong21@mails.ucas.ac.cn

Shichen Wu
WeBank Blockchain Team
Shenzhen, China
shichenw@mail.sdu.edu.cn

Xiuzhen Cheng
Shandong University
Qingdao, China
xzcheng@sdu.edu.cn

Abstract

Rotating leader Byzantine fault-tolerant (BFT) protocols, also known as protocols in the leader-speaks-once (LSO) model, have become a topic of interest with the rise of blockchains. Many recent efforts have been made to lower the latency and simplify the design. However, existing works often sacrifice expected latency (latency when failures might occur) to enjoy higher good-case latency.

In this work, we present a generic approach to building rotating leader BFT protocols that are both simple and fast. We present two variants: HumBird-1, a synchronous BFT under the $f < n/2$ assumption; HumBird-2, a partially synchronous BFT under the $f < n/3$ assumption, where n is the number of replicas in the system and f is the number of Byzantine failures. We show that by relaxing LSO to the leader-speaks-twice (LST) model and introducing a special uncle block field, we can significantly reduce both good-case latency and expected latency. Our experimental results show that compared to the state-of-the-art rotating leader BFT, HumBird-1 and HumBird-2 achieve up to 32% lower latency and 70% higher throughput, and 95% lower latency and 140% higher throughput, respectively.

Keywords

Byzantine Fault Tolerance, Rotating Leader, Atomic Broadcast, Reorg Resilience, Leader-speaks-twice Model

1 Introduction

Byzantine fault-tolerant (BFT) protocols are fundamental to practical distributed systems such as blockchains. Conventional synchronous BFT [2, 5, 6, 21] (where there exists a known upper bound Δ on message transmission and processing) and partially synchronous BFT [19, 32, 33, 57] (there exists an unknown bound [34]) protocols are leader-based where the leader proposes new blocks and collects the votes from other replicas. Historically, most BFT protocols assume a *stable leader* where a leader continues to propose new blocks in the *normal case protocol*. When the current leader is suspected to be faulty, a *view change* process is triggered to elect a new leader [2, 19, 32, 33]. In stable leader BFT, one can optimize the normal case and throw some expensive procedures to the view change, so the system achieves high performance most of the time. The drawback is that a slow or malicious leader can degrade the system performance significantly and affect the fairness of the transactions [25, 31, 39, 58].

Rotating leader BFT. In recent years, the rise of blockchain makes it important to design the *rotating leader* BFT, also known as protocols in the leader-speaks-once (LSO) model [36]. In the LSO model, each leader proposes a new block and is immediately rotated. Such a design can mitigate the issues caused by a slow or malicious leader. The rotating leader paradigm can be viewed as a protocol in which the view change process is triggered every time when a new leader is elected. Thus, one can not easily throw expensive procedures to the view change and should alternatively build a more *balanced* protocol with roughly the same expected performance in the normal case protocol and during view changes. Many recent rotating leader BFT protocols explore new designs in this way. Notable examples include SyncRR [6] in the synchronous setting, and Simplex [20] and DispersedSimplex [50] in the partially synchronous setting.

Rotating leader BFT from the perspective of Proof-of-Authority (PoA)/Proof-of-Stake (PoS) protocols. Rotating leader BFT is conceptually similar to PoA/PoS protocols used by cryptocurrencies such as Gasper by Ethereum 2.0 [15] (with a market capitalization of $519 billion), BNB Smart Chain by Binance (with a market capitalization of $121 billion), and Polkadot (with a market capitalization of $43 billion)[1]. In these systems, the protocol is often logically divided into a *longest chain* sub-protocol and a *finality gadget* sub-protocol. The longest chain sub-protocol is largely inspired by Nakamoto consensus [45], in which replicas take turns to propose new blocks based on their local views of the *longest chain* [38] or the *heaviest chain* [14]. Since the local views of honest replicas might not be consistent, forks of chain can be formed. To make sure that honest replicas converge to the same chain, the finality gadget uses a vote-based mechanism similar to conventional BFT to allow blocks to be *finalized*. Namely, once a block is finalized, its position in the chain will never be reversed.

The research question. Most PoA and PoS protocols directly embed the finality rules into the longest chain sub-protocol and thus do not have an explicit *view change* process. Besides, to our knowledge, most PoA/PoS protocols only have two main message types: Propose and Vote, and each event is triggered periodically based on

*Sisi Duan and Minghui Xu are the corresponding authors.

[1] Data source (accessed in Sep 2025): https://coinmarketcap.com/currencies.

Table 1: Comparison of synchronous rotating-leader BFT.

Protocol	$f <$	Good-case latency[†]	Expected latency[†]	# Msg types	Communication[★]	Reorg resilience
Shi's protocol [49]	$n/3$	$O(n\Delta)$	$O(n\Delta)$	1	$O(Ln^2)$	✗
Apollo [10][§]	$n/2$	$O(\gamma\delta)$[§]	$O(\gamma\Delta)$[§]	3	$O(Ln^2+\kappa n^3)$	✓
Sync Streamlet [21][◇]	$n/2$	12Δ	1016Δ[◇]	2	$O(Ln^3)$	✗
OptSync [51][‡]	$n/2$	$3\Delta+\delta$ / 3δ (opt.)[‡]	$11\Delta+4\delta$	8	$O(Ln^2+\kappa n^3)$	✓
RandPiper SMR [12]	$n/2$	$4\Delta+3\delta$	$17\Delta+3\delta$	4	$O(Ln+\kappa n^2\log n)$	✓
RL-SHS [2][★★]	$n/2$	$2\Delta+\delta$	$11\Delta+4\delta$	4	$O(Ln^2+\kappa n^3)$	✓
SyncRR [6]	$n/2$	$2\Delta+2\delta$	$10.5\Delta+3\delta$	3	$O(Ln^2+\kappa n^3)$	✓
HumBird-1 (this work)	$n/2$	2Δ	7.5Δ	2	$O(Ln^2+\kappa n^3)$	✓
HumBird-1 + State transfer (this work)	$n/2$	2Δ	7.5Δ	3	$O(Ln+\kappa n^3)$	✓

n is the number of replicas, f is the maximum number of faulty replicas, Δ is the upper bound on network transmission and processing, and δ is the actual network delay. [†] See Sec. 2 for the formal definition of latency metrics. Informally, the good-case latency is the minimum time for all honest replicas to finalize a block proposed by an honest leader. The expected latency is the expected time between the submission of a transaction and its finalization by all honest replicas. We compute the expected latency of all protocols under the setting where leaders are randomly elected. [★] To make a fair comparison, we report the communication complexity by assuming digital signatures only, i.e., a certificate has $O(n)$ digital signatures. [§] γ is a security parameter of Apollo. Apollo ensures that any sequence of γ rounds contains at least one honest leader. [◇] Sync Streamlet requires eight consecutive honest leaders to achieve finality. This requirement significantly increases the protocol's expected latency, as the probability of electing eight consecutive honest leaders is low under random leader rotation. See Appendix F for analysis. [‡] OptSync can achieve a good-case latency of 3δ when the number of faulty replicas is less than $n/4$. [★★] We use RL-SHS to denote the rotating-leader version of Sync HotStuff.

Table 2: Comparison of partially synchronous rotating-leader BFT. All protocols assume $f < n/3$.

Protocol	Good-case latency[†]	Expected latency[†]	# Msg types	Communication	Reorg resilience[†]
Partially Sync Streamlet [21][◇]	4Δ	33.56Δ[◇]	2	$O(Ln^3)$	✗
Chained HotStuff [57][□]	7δ	decided by Pacemaker[□]	3	$O(Ln^2+\kappa n^2)$	✗
SSF [35][★]	$3\Delta+\delta$	$5\Delta+\delta$	3	unbounded[★]	✓
ICC0 [18]	3δ	$1.5\Delta+5.5\delta$	5	$O(Ln^2+\kappa n^3)$	✗
Commit Moonshot [30]	3δ	$1.5\Delta+3.5\delta$	8	$O(Ln^2+\kappa n^3)$	✗
Simplex [20][‡]	3δ	$1.5\Delta+3.5\delta$	3	unbounded[‡]	✓
DispersedSimplex [50]	3δ	$1.5\Delta+3.5\delta$	7	$O(Ln+\kappa n^3)$	✓
HumBird-2 (this work)	3δ	$3.11\Delta+3\delta$	3	$O(Ln^2+\kappa n^2)$	✓
HumBird-2 + State transfer (this work)	3δ	$3.11\Delta+3\delta$	4	$O(Ln+\kappa n^2\log n)$	✓

[†] In the partially synchronous model, we consider good-case latency, expected latency, and reorg resilience of all protocols after GST (i.e., after the network becomes synchronous). [◇] Partially Sync Streamlet requires five consecutive honest leaders to achieve finality. We provide the full analysis of its expected latency in Appendix F. [□] Chained HotStuff uses a module called Pacemaker to trigger view change. The expected latency of the protocol depends on the specific implementation of the Pacemaker. [‡] Simplex requires each proposal to include its notarized parent blockchain, making the size of each proposal proportional to the size of the blockchain. [★] SSF requires each proposal to include the current message log of the leader, causing the size of each proposal to grow unboundedly over time.

local physical clocks. Such simplification makes it arguably easier to implement and debug. This is in sharp contrast to existing rotating leader BFT protocols, where the view change rules are usually quite complicated and replicas may need to wait for timeouts or messages from other replicas before entering a new view. Even for a synchronous rotating leader BFT protocol, e.g., SyncRR [6], the view change process lasts for $7\Delta+\delta$ in the worst case, in contrast to 2δ in the best case, where δ denotes the actual network delay. Unfortunately, existing PoA and PoS protocols suffer from various attacks and most of them are not provably secure yet [38, 40, 46, 59, 60].

Thus, an open research question is:

Can we build a secure rotating leader BFT protocol that is both simple and fast?

Our approach: rotating leader BFT in the leader-speaks-twice (LST) model. We show that by slightly relaxing the LSO model to the leader-speaks-twice model (i.e., each leader proposes two blocks and is then rotated), we can build a secure protocol that is both simple and fast. Our insight is that we can reduce the correctness of view change in the rotating leader protocol to the correctness of *reorg resilience* [26, 61], i.e., a block proposed by an honest leader will eventually be finalized. We show that reorg resilience can be achieved when the network is synchronous and show the following results. We also summarize our results in Table 1 and Table 2.

- We present HumBird-1, a synchronous BFT protocol under the $f < n/2$ assumption, where f is the maximum number of Byzantine replicas and n is the total number of replicas in the system. HumBird-1 only has two message types and 2Δ good-case latency. Compared to the state-of-the-art synchronous rotating leader BFT, i.e., SyncRR, HumBird-1 reduces the worst-case duration of view change from $7\Delta+\delta$ to 4Δ and decreases the expected latency from $10.5\Delta+3\delta$ to 7.5Δ. By slightly revising the protocol and having three message types, the communication complexity of HumBird-1 can be further lowered.
- We abstract away some techniques in HumBird-1 and generalize it to HumBird-2, a partially synchronous BFT under the

$f < n/3$ assumption. HumBird-2 achieves reorg resilience when the network is synchronous and the conventional security properties under the partially synchronous assumption. By doing so, HumBird-2 achieves 3δ good-case latency and $3.11\Delta + 3\delta$ expected latency. Furthermore, HumBird-2 also achieves lower communication complexity than the state-of-the-art protocols.

Summary of contributions. Our work makes the following contributions.

- We show that one can build a rotating leader BFT protocol that is both simple and fast in the leader-speaks-twice model, and we provide some generic solutions to build provably secure protocols.
- We present HumBird-1, a synchronous BFT protocol that achieves both lower good-case latency and expected latency.
- We present HumBird-2, a partially synchronous BFT protocol that not only achieves the same good-case latency as the state-of-the-art protocol, but also enjoys lower communication complexity.
- We implement HumBird-1, HumBird-2, and three known rotating leader BFT protocols. Our experimental results show that compared to the state-of-the-art protocol, HumBird-1 achieves 20% lower latency in failure-free cases, and achieves 32% lower latency and 70% higher throughput under failures. We also evaluate our HumBird-2 protocol in failure-free cases and show that HumBird-2 achieves up to 95% lower latency and 140% higher throughput than the baseline.

2 System Model and Building Blocks

Byzantine fault tolerance (BFT). In a BFT protocol, clients *submit* transactions (requests) and replicas *finalize* them. The client obtains a final response to the submitted transaction from the responses it receives from replicas. We consider a BFT system with n replicas, $\{p_1, p_2, \ldots, p_n\}$, among which at most f replicas may fail arbitrarily. These faulty replicas are also called Byzantine failures and non-Byzantine replicas are called *honest* replicas.

We use the definition of atomic broadcast below to define correctness. Atomic broadcast is often used interchangeably with BFT. Informally, the main difference is that ABC does not involve the role of the clients.

- **Safety**: If an honest replica *finalizes* a message m before it *finalizes* m', then no honest replica *finalizes* the message m' without first *finalizing* m.
- **Liveness**: If an honest replica *proposes* a message m, then all honest replicas eventually *finalize* m.

Network assumption. We consider both a synchronous network and a partially synchronous network [34]. In the synchronous network, there exists a known upper bound Δ for message processing and transmission delay. In the partially synchronous network, there exists an unknown upper bound for message processing and transmission. An alternative notion of the partially synchronous model is that there exists an unknown global stabilization time (GST) such that after GST, messages sent between two honest replicas arrive within a fixed delay Δ. The two notions are used equivalently and we follow the latter one to prove the correctness of our partially synchronous protocol. For the partially synchronous network, we assume that replicas have synchronized clocks after GST. The same assumption is reasonable [2], and made by many existing consensus protocols [3, 21, 35, 49].

We consider $n \geq 2f+1$ setting for our synchronous protocol and $n \geq 3f+1$ setting for our partially synchronous protocol. Note that both bounds are optimal for the corresponding timing assumption. Without loss of generality, we assume $n = 2f + 1$ and $n = 3f + 1$ to simplify our presentation.

Chain of blocks. We use *block* B to denote a block. The first block in the system is called the *genesis* block B_0. Each block (except for B_0) has a *parent block*, i.e., the previous block in the chain. We say that a block B *extends* its parent block B'. We may also say that block B extends B', if B' is on the chain led by the parent block of B. Any block extended by B is called an *ancestor* of B. Two blocks B and B' are conflicting if neither of them extends the other. Blocks B and B' are equivocating blocks of each other if they are conflicting and are proposed by the same leader in the same slot (the notion of *slots* will be introduced later).

Malicious reorganization attack (reorg attack for short) [47, 48] denotes an attack in which the proposed blocks by honest leaders are reorganized and discarded. We borrow the *reorg resilience* notion [26, 28, 35, 61] in our work and build reorg-resilient protocols. The property is defined as follows.

- **Reorg resilience**: If an honest leader proposes a block B in the synchronous network, all honest replicas will eventually finalize B.

To our knowledge, reorg resilience can only be achieved when the network is synchronous. This is mainly because a Byzantine leader can cause a malicious reorganization by withholding its block and releasing the block later. When the network is asynchronous, one can never differentiate a faulty leader from a slow leader. Thus, it is extremely challenging, if not impossible, to build reorg-resilient protocols.

Uncle blocks. We define another concept called uncle blocks. Some block B may reference an uncle block that might be conflicting with B's parent block. Once B is delivered, the chains led by B's parent block and B's uncle block are merged and finalized simultaneously.

Byzantine quorums and quorum certificates. A *Byzantine quorum* (or *quorum* in short) denotes certain number of replicas. In the $n = 2f + 1$ setting (i.e., our HumBird-1 protocol), a Byzantine quorum consists of $f + 1$ replicas. In the $n = 3f + 1$ setting (i.e., our HumBird-2 protocol), a Byzantine quorum consists of $2f + 1$ replicas. *Votes* for a block from a quorum are called a *quorum certificate*, or *certificate* for short. A block with a valid certificate is called a *certified block*.

Slots. Our protocols proceed in slots. Each slot t begins at time τ_t and lasts for 2Δ time, i.e., $\tau_{t+1} - \tau_t = 2\Delta$. We use the $slot()$ function to denote the slot number of a certificate or a block. For example, if cer is a certificate for block B, $slot(\text{cer})$=$slot(B)$.

Notations. Every replica maintains a block tree $\mathcal{T}$ to store received blocks. For each block B in $\mathcal{T}$, every replica maintains a vote set $V(B)$ to record received votes for B.

Latency metrics. We follow existing literature [6, 20, 27, 56] to define the latency metrics. In particular, we define the finalization

[2] Detailed discussion can be found at https://decentralizedthoughts.github.io/2026-03-07-simultaneous-agreement/

latency of a transaction as the duration between the time the transaction is submitted to and the time the transaction is finalized by all honest replicas. Note that for partially synchronous protocols, latency is measured after GST. We assume that each transaction is submitted to all honest replicas right before a new leader proposes the next block. In this way, the good-case latency is the minimum finalization latency of all submitted transactions under f failures, and the expected latency is the average finalization latency of all submitted transactions under f failures.

Cryptographic algorithms. We use digital signatures, making the public-key infrastructure (PKI) assumption. We use $\langle \mu \rangle_i$ to denote a message μ signed by replica p_i and use $\langle \mu \rangle_*$ to denote one signed by any replica. We assume a cryptographic collision-resistant hash function denoted as $H(\cdot)$. We use Merkle tree as the cryptographic accumulator. Given a set of values, the Merkle tree root serves as an accumulation value for a set, and each Merkle tree branch is a witness for one value in the set.

Erasure coding scheme. An (m, n) erasure coding scheme over an alphabet $\mathcal{M}$ is a pair of algorithms (*encode*, *decode*), where *encode*: $\mathcal{M}^m \rightarrow \mathcal{M}^n$ and *decode*: $\mathcal{M}^m \rightarrow \mathcal{M}^m$. The *encode* algorithm takes as input a data block, consisting of m data fragments, and outputs $n > m$ coded fragments. The *decode* algorithm takes as input any m-size subset of coded fragments and outputs the original data block containing m data fragments. Namely, if $[d_1, \ldots, d_n] \leftarrow encode(\mathcal{M})$, then $decode(d_{i_1}, \ldots, d_{i_m}) = \mathcal{M}$ for any distinct $i_1, \ldots, i_m \in [1..n]$. In this paper, we use the Reed-Solomon code to realize an (m, n) erasure coding scheme, where each coded fragment has a size of $\frac{L}{m}$ if the input block is of size L.

3 Motivation and Overview of HumBird

3.1 A Recap of Existing Rotating Leader BFT

Rotating leader BFT from the perspective of PoA/PoS. In this paper, we borrow the notations from PoA and PoS protocols to describe rotating leader BFT. As mentioned in the introduction, PoA and PoS protocols often consist of two sub-protocols: a longest chain variant sub-protocol where replicas propose blocks; a finality gadget where replicas vote for the blocks and eventually finalize some blocks. Without loss of generality, we consider a simplified model where each slot t consists of two phases and all replicas vote, as summarized below. Let τ_t be the time when slot t begins.

Propose. At time τ_t, query LeaderElection to determine whether p_i is the designated leader. If so, p_i queries ForkChoice and obtains head, i.e., a block from its block tree $\mathcal{T}$. It then creates a block B that extends head and sends B to all replicas. Upon receiving B, each replica adds B to $\mathcal{T}$.

Vote. At time $\tau_t + \Delta$, each replica p_i checks whether it has received a proposed block in slot t from the leader. If p_i has received a proposed block B, it queries ForkChoice to obtain some information from $\mathcal{T}$, and verifies whether B is a valid block. If so, p_i sends a vote on B to all replicas. After receiving a vote, each replica adds the vote to the block tree $\mathcal{T}$. Let $V(B)$ be the set of received votes for block B. If the size of $V(B)$ reaches a pre-specified threshold, the status of B is changed accordingly.

The LeaderElection and ForkChoice functions are defined differently in different systems. For instance, in Gasper [15], the leader is selected randomly from all replicas and ForkChoice selects the *heaviest* chain from the block tree, where a chain is in general heavier if it consists of more blocks and more votes. For BNB Smart Chain [38], the leaders are selected in a round-robin manner. The ForkChoice function also selects the heaviest chain following similar rules to Gasper.

One main issue of PoA and PoS protocols is the so-called *malicious reorganization* attacks. As shown in Figure 1, malicious reorganization denotes an attack in which a Byzantine leader delays its block proposal. After the block is released, a block proposed by an honest leader does not belong to the heaviest chain and will be *reorganized*. Malicious reorganization attacks can create many issues such as penalties of honest replicas [41, 59, 61] and even liveness failures [38, 46, 62]. Some recent works such as SSF [35] seek to address the issue, but suffer from high communication cost, as summarized in Table 2.

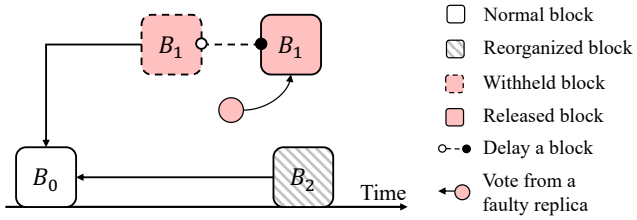


Figure 1: An example of a reorganization attack. Let block B_0 be a block known by all replicas. In the next slot, a Byzantine leader p_i withholds its block B_1. The next leader is honest and is not aware of B_1. It creates a block B_2 that extends B_0. Right before B_2 is released, since p_i knows exactly when B_2 will be released, p_i releases B_1. After receiving B_1, the chain led by B_1 becomes the canonical chain as it is heavier (with more votes). The block B_2 is then reorganized and discarded.

The relationships between ForkChoice and safe view change in rotating leader BFT. Although conventional BFT often do not describe the protocol using the above paradigm, we emphasize that a *safe* view change in rotating leader BFT corresponds to a well-defined ForkChoice function, which is why conventional BFT protocols do not suffer from malicious reorganization attacks.

The safe view change process, however, is not cheap. In fact, even in synchronous BFT, the process is often made responsive, where the new leader often has to wait for a sufficiently large number of messages before it starts a new view. Also, the communication cost and latency of such a process become larger. We use Sync HotStuff [2] as an example to discuss the claim in detail in Appendix A. In fact, the challenges of balancing normal case protocol and view change can also be reflected in the good-case latency and expected latency. As shown in Table 1, such an approach to building rotating leader BFT can make the expected latency very large.

Remark on view vs. fixed view schedule. We would like to further clarify the nuance between the view and the fixed view schedule. Conventional BFT protocols [19, 57] often use the term view change to denote the change of the leader, but the leader is not assumed to change frequently. In most such protocols, replicas discard all messages from earlier views after entering a new view.

In rotating leader BFT, especially many recent works [3, 21, 27, 42, 44, 56], the view number is often used to denote the unique leader, and the leader is rotated periodically, either based on physical clocks or on some pre-determined rules. Such protocols are also called protocols with a *fixed* view schedule. In most such protocols, replicas keep the messages from prior views. To make it clear, we use the notion of *slots* from PoA/PoS to denote the leaders, and thus replicas can keep messages from prior "views".

3.2 Overview of HumBird

We address the above issues and propose two rotating-leader BFT protocols: HumBird-1, a synchronous BFT protocol assuming $n \geq 2f+1$, and HumBird-2, a partially synchronous BFT protocol assuming $n \geq 3f+1$. Our HumBird protocols do not suffer from the issues above by not having a specific view change sub-protocol. Our protocols proceed in *slots*, where each slot lasts for 2Δ time. We show that a safe view change can be reduced to *reorg resilience*, and protocols can be made much more efficient (with both low good-case latency and expected latency) in this way.

A crucial technique in our protocols is by relaxing the leader-speaks-once (LSO) model to the leader-speaks-twice (LST) model, where each leader can propose two blocks in two consecutive slots. We also slightly change the proposal and vote phases as follows.

Propose. At time τ_t, the leader proposes a block by extending the *highest certified block* in its block tree $\mathcal{T}$, where the highest certified block denotes a *safe* branch in the block tree.

Vote. At time $\tau_t + \Delta$, each replica casts its vote according to ForkChoice. Each replica might also include its highest certified block in its vote, a process similar to the view change in conventional rotating leader BFT.

The LST model can enable a *safe view change* in a more efficient way. We show an example in Figure 2. Let t_2 be the slot of the first slot in which p_2 is the leader. If p_2 fails to receive the highest certified block at the beginning of slot t_2 (an odd slot), it may choose to create B_2 that extends B_0, and B_2 is conflicting with the block output by p_3's ForkChoice (a replica will not vote for a block conflicting with the output of ForkChoice). In our protocol, p_3 forwards its highest certified block to p_2 in slot t_2. Therefore, p_2 is guaranteed to receive the highest certified block B_1 before slot t_2+1 (an even slot), i.e., p_2 has already completed a *safe view change* and will create a block proposal B_3 that will be voted for by all honest replicas in an even slot.

A new challenge arises: since a safe view change is not completed in slot t_2 (an odd slot), the block proposed in slot t_2 might still be reorganized. To address this issue, we additionally define a *uncle* field. In an even slot, the leader may use the uncle field to carry the block of the odd slot. In this way, both blocks can be finalized simultaneously, and our protocol is reorg-resilient.

We show that by using the LST model and the uncle field design, we can build a synchronous BFT and a partially synchronous BFT that are both simple and fast.

Remark on forwarding the highest certified blocks. Forwarding the highest certified blocks to the next leader is a common *trick* in conventional BFT protocols [1, 2, 19, 57]. However, even with such a forwarding process, the view change process can again be enlarged. We borrow this process to our protocols to build reorg-resilient protocols. Our key insight is that the workflow of achieving reorg resilience (via the ForkChoice function) can be made cheaper than the safe view change process in conventional BFT. The expected latency can thus be made low without sacrificing good-case latency.

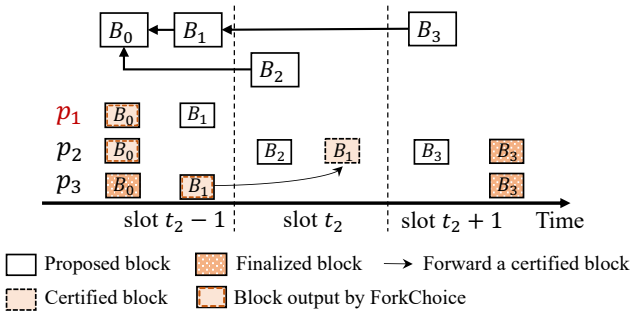


Figure 2: Illustration of HumBird. B_0 is the highest certified block known to p_1, p_2, and p_3 when they enter slot $t_2 - 1$. In slot $t_2 - 1$, the Byzantine leader p_1 proposes a block B_1 extending B_0, but only p_3 receives the certificate for B_1. When the replicas enter slot t_2, ForkChoice of p_1 and p_2 outputs B_0, while p_3's ForkChoice outputs B_1. The next leader p_2 is not aware of B_1 and proposes a block B_2 extending B_0. Instead of voting for B_2, p_3 forwards the certificate for B_1 to p_2. Therefore, p_2 proposes B_3 extending B_1 in slot $t_2 + 1$, which will be voted for by all honest replicas.

4 HumBird-1: Synchronous Rotating Leader BFT

We present HumBird-1, a synchronous rotating leader BFT protocol assuming $n \geq 2f+1$. In this section, we first present HumBird-0, a warm-up protocol that is both safe and live, but not reorg-resilient. We then build up on it and show how to achieve reorg resilience.

4.1 HumBird-0: A Warm-up Protocol

We show the pseudocode of HumBird-0 in Algorithm 1. HumBird-0 does not consider uncle blocks and mainly uses the LST model to define the rules. Each leader *owns* two consecutive slots: an odd slot and an even slot.

Fork choice rule. We now formally define ForkChoice. ForkChoice selects at most two branches from the block tree and outputs a tuple (head, candidate, cer_c), where head and candidate are blocks, and cer_c is a certificate. The chain led by head is called the canonical chain of a replica. The chain led by candidate is a branch that can be safely extended to create new blocks. cer_c is a certificate for candidate. In slot t, the function outputs the tuple by the following rules.

- If slot t is an odd slot, ignore the votes that are received after time $\tau_t + \Delta$. If slot t is an even slot, ignore all votes received after $\tau_{t-1} + \Delta$. Then, the certified block with the highest slot number in the block tree becomes head.
- Prune the block tree by removing the blocks proposed before $slot(\text{head})$. The certified block B with the highest slot in the

Algorithm 1 The main protocol of HumBird-0. Code for p_i.

1: **Global parameter:** slot t
2: **Local parameters:** a block tree $\mathcal{T}$, a vote set $V(B)$ for each block B in $\mathcal{T}$
3: Initialize the following parameters:
4: $\mathcal{T} \leftarrow \emptyset, t \leftarrow 1, \tau_1 \leftarrow 0$
▷ For the leader of slot t:
5: **upon** a slot t begins at time τ_t **do**
6: let (head, candidate, cer_c) be ForkChoice's output
7: create block $B = \langle \text{vals}, H(\text{candidate}), t \rangle_i$, where vals consists of a batch of transactions
8: send message $m = \langle \text{PROPOSE}, B, t, \text{cer}_c \rangle_i$ to all replicas
▷ For all replicas:
9: **upon** receiving the first valid $\langle \text{PROPOSE}, B, t, \text{cer} \rangle_j$ from the leader p_j of slot t before $\tau_t + \Delta$ **do**
10: wait until $\tau_t + \Delta$
11: let (head, candidate, cer_c) be ForkChoice's output
12: **if** cer is a certificate for the parent of B **and**
13: $slot(\text{cer}) \geq slot(\text{head})$ **then**
14: forward the $\langle \text{PROPOSE} \rangle$ message to all replicas
15: send vote $v = \langle \text{VOTE}, t, H(B) \rangle_i$ to all replicas
16: **else**
17: send cer_c to the leader p_j
18: **upon** receiving a valid $\langle \text{PROPOSE}, B, t, \text{cer} \rangle_j$ in slot t **do**
19: $\mathcal{T} \leftarrow \mathcal{T} \cup \{B\}$
20: **upon** receiving a vote $v = \langle \text{VOTE}, t', h' \rangle_*$ **do**
21: wait until the block B' corresponding to v is added to $\mathcal{T}$
22: $V(B') \leftarrow V(B') \cup \{v\}$
23: **upon** an even slot t ends at $\tau_t + 2\Delta$ **do**
24: **if** $\mathcal{T}$ contains only one block B of slot t such that $|V(B)|$
25: $\geq f + 1$ **and** no equivocation is detected **then**
26: send $V(B)$ to all replicas
27: **if** B has not been finalized **then**
28: **finalize** B and all its ancestors

pruned block tree becomes the candidate. If multiple blocks have the highest slot number, break the tie by choosing an arbitrary one as candidate. Then, the certificate for candidate is cer_c.

The workflow. As is shown in Algorithm 1, when slot t begins at τ_t, the leader queries ForkChoice and obtains the tuple (head, candidate, cer_c). It then creates a block B that extends block candidate, and sends B and cer_c to all replicas. Upon receiving a block proposal (B, cer), each replica p_i also queries ForkChoice and obtains a tuple (head, candidate, cer_c). It verifies whether cer in the proposal is a valid certificate and $slot(\text{cer}) \geq slot(\text{head})$. If so, p_i creates a vote on B and sends it to all replicas. Otherwise, p_i sends cer_c to the leader. Any vote or certificate received by a replica is added to its block tree. Note that a certificate is a set of votes and is processed in the same way as votes.

If t is an even slot, at the end of slot t, each replica p_i begins the finalization rule. Specifically, if there exists only one block B proposed in t with $f + 1$ matching votes and no equivocation is detected, p_i sends $V(B)$, i.e., a certificate on B, to all replicas. Then, p_i finalizes B and all its ancestors.

Remark. Detecting equivocation before finalizing a block is a common approach to ensuring safety in synchronous BFT [1, 2, 4, 6]. We follow this approach to build safe synchronous protocols. However, this approach alone is not sufficient to ensure liveness. In conventional synchronous protocols, replicas run an additional view change process after finalizing a block. As mentioned in the overview, running a view change whenever the leader is rotated can be expensive. HumBird-0 can be viewed as protocol with a cheaper but eventually safe view change.

Sketch of Correctness. We begin with safety. Towards a contradiction, suppose that an honest replica p_i finalizes B in slot t, and another honest replica p_j finalizes B' in slot t such that $B \neq B'$. According to the protocol, p_i has received $f + 1$ matching votes on B and p_j has received $f + 1$ matching votes on B'. As there are at most f faulty replicas, at least one honest replica casts a vote on B and at least one honest replica casts a vote on B'. In each vote message, each replica includes the block proposal. Thus, both p_i and p_j should receive both B and B'. According to our protocol, honest replicas will not finalize a block if an equivocation is detected, a violation. Therefore, conflicting blocks can not be both finalized in a slot. Additionally, when an honest replica finalizes a block B in slot t, each honest replica will receive a certificate for B and ForkChoice will output head = B. This ensures that all honest replicas will never vote for any block conflicting with B. Therefore, any blocks finalized after slot t must extend B and safety holds.

For liveness, we mainly need to prove that given a block B proposed by an honest leader p_j, all honest replicas will vote for B. We show that this is the case for an even slot t. In our protocol, each replica creates a vote on a proposal $\langle \text{PROPOSE}, B, t, \text{cer} \rangle_j$, if $slot(\text{cer}) \geq slot(\text{head})$, where head is the output of ForkChoice. Let C be a set of certificates received by honest replicas before they vote in an odd slot owned by p_j. According to our protocol, each replica will send its certificate to p_j in the odd slot if the replica has a certificate with a higher slot than $slot(\text{cer})$. Thus, the cer_c included by p_j's block proposal in the even slot is the highest in C. All honest replicas will satisfy $slot(\text{cer}) \geq slot(\text{head})$ and cast a vote in the even slot. Liveness thus holds.

Note that if B is proposed in an odd slot, it might be reorganized. Although liveness still holds, the good-case latency is quite high (as transactions submitted to an honest leader can only be finalized in the next even slot). We will address this issue shortly in HumBird-1.

4.2 HumBird-1: Reorg-Resilient Synchronous BFT

We build on top of HumBird-0 to make the protocol reorg-resilient and we call our protocol HumBird-1. To prevent a block proposed in an odd slot from being reorganized, we use a *uncle* block field to include such a block. As shown in Algorithm 2, we highlight the differences in pink . In addition, we also provide a fast path that can finalize a block in 2Δ time. We highlight the modifications for the fast path in gray.

Reorg resilience. As shown in HumBird-0, the block proposed by an honest leader in the even slot is guaranteed to be finalized. However, the block proposed in the odd slot might be reorganized. We thus require the block B proposed in the even slot to include the block from the odd slot as an uncle block (lines 8-9). Upon receiving a block with an uncle block, each replica verifies whether the uncle block is well-formed (line 18). When B is finalized, all the blocks

Algorithm 2 The main protocol of HumBird-1. Code for p_i.

1: **Global parameter:** slot t
2: **Local parameters:** a block tree $\mathcal{T}$, a vote set $V(B)$ for each block B in $\mathcal{T}$
3: Initialize the following parameters:
4: $\quad \mathcal{T} \leftarrow \emptyset, t \leftarrow 1, \tau_1 \leftarrow 0$
▷ For the leader of slot t:
5: **upon** a slot t begins at time τ_t **do**
6: $\quad$ let $(\text{head}, \text{candidate}, \text{cer}_c)$ be ForkChoice's output
7: $\quad$ set the commit flag cf as 0 and uncle as $\perp$
8: $\quad$ **if** slot t is an even slot **and** $slot(\text{cer}_c) < t-1$ **then**
9: $\quad\quad$ let uncle be the block p_i proposes in slot $t-1$
10: $\quad$ **if** slot t is an even slot **or** $slot(\text{cer}_c) = t-1$ **then**
11: $\quad\quad$ $cf \leftarrow 1$
12: $\quad$ create $B = \langle \text{vals}, H(\text{candidate}), t, cf, \text{uncle} \rangle_i$
13: $\quad$ send message $m = \langle \text{PROPOSE}, B, t, \text{cer}_c \rangle_i$ to all replicas
▷ For all replicas:
14: **upon** receiving the first valid $\langle \text{PROPOSE}, B, t, \text{cer} \rangle_j$ from the leader p_j of slot t before $\tau_t + \Delta$ **do**
15: $\quad$ wait until $\tau_t + \Delta$
16: $\quad$ let $(\text{head}, \text{candidate}, \text{cer}_c)$ be ForkChoice's output
17: $\quad$ **if** cer is a certificate for the parent of B **and** $slot(\text{cer}) \geq slot(\text{head})$
18: $\quad\quad$ **and** the uncle block of B is valid **then**
19: $\quad\quad\quad$ forward the $\langle \text{PROPOSE} \rangle$ message to all replicas
20: $\quad\quad\quad$ send vote $v = \langle \text{VOTE}, t, H(B) \rangle_i$ to all replicas
21: $\quad$ **else**
22: $\quad\quad\quad$ send cer_c to the leader p_j
23: **upon** receiving a valid $\langle \text{PROPOSE}, B, t, \text{cer} \rangle_j$ in slot t **do**
24: $\quad$ $\mathcal{T} \leftarrow \mathcal{T} \cup \{B\}$
25: **upon** receiving a vote $v = \langle \text{VOTE}, t', h' \rangle_*$ **do**
26: $\quad$ wait until the block B' corresponding to v is added to $\mathcal{T}$
27: $\quad$ $V(B') \leftarrow V(B') \cup \{v\}$
28: **upon** a slot t ends at $\tau_t + 2\Delta$ **do**
29: $\quad$ **if** $\mathcal{T}$ contains only one block B of slot t such that
30: $\quad\quad$ $|V(B)| \geq f+1$ **and** no equivocation is detected **and**
31: $\quad\quad$ the commit flag of B is 1 **then**
32: $\quad\quad\quad$ send $V(B)$ to all replicas
33: $\quad\quad\quad$ **if** B has not been finalized **then**
34: $\quad\quad\quad\quad$ **for** each unfinalized ancestor of B **do**
35: $\quad\quad\quad\quad\quad$ **finalize** the ancestor and its uncle block
36: $\quad\quad\quad\quad$ **finalize** B and the uncle block of B

on the chain led by B and their corresponding uncle blocks are finalized (lines 35-36).

Fast path. We can enable a fast path for HumBird-1 to achieve low good-case latency. Specifically, when an honest leader creates a new block B in a slot t, it introduces a commit flag cf into the block to help determine whether the block can be finalized at $\tau_t + 2\Delta$. The commit flag of B will be set as 1 if and only if one of the following conditions holds (line 10):

- slot t is an even slot ;
- $slot(\text{cer}_c) = t-1$, where cer_c is the output of ForkChoice.

Otherwise, the commit flag of B will be set as 0 (line 7). Only blocks with a commit flag of 1 can be finalized at $\tau_t + 2\Delta$ (line 31). Additionally, the fork choice rule ForkChoice also needs to be slightly modified. Namely, in slot t, if a certified block B of slot $t-1$ is received before $\tau_t + \Delta$ and the commit flag of B is 1, output head as B. Otherwise, apply the same fork choice rule as that in HumBird-0 to output head, candidate, and cer_c.

Random leader election. We can also implement the Leader-Election function with a random leader election method. This can be implemented via the common coin protocol [16] or verifiable random function (VRF) [43] without affecting the correctness.

Latency. In the good case, the latency of HumBird-1 is 2Δ. This is mainly because all replicas in each slot t can receive a valid certificate for the block of slot t. Thus, the leader in each slot $t+1$ will set the commit flag of its block as 1. Therefore, any block proposed at the beginning of a slot must be finalized at the end of the slot. Otherwise, the block proposed in an odd slot might not be finalized in 2Δ time. However, it is guaranteed to be finalized by the end of the next even slot. The latency is thus 4Δ. When there are $\lfloor \frac{n-1}{2} \rfloor$ faulty replicas and HumBird-1 uses random leader election, the protocol has an expected latency of 7.5Δ. We provide the full analysis of its expected latency in Appendix F.

Communication complexity. HumBird-1 achieves $O(Ln^2 + \kappa n^3)$ communication, where L is the size of a block. The Ln^2 term is because every replica forwards its received block to all replicas. The κn^3 term is because every replica forwards the certificate (of size $O(\kappa n)$) for the finalized block to all replicas.

We show the correctness of HumBird-1 in Appendix C.

4.3 Communication-Efficient HumBird-1

We can further improve the communication complexity of HumBird-1 from $O(Ln^2 + \kappa n^3)$ to $O(Ln + \kappa n^3)$. Recall that the bottleneck of HumBird-1 is that every replica forwards its received block proposal to all replicas. We can optimize it using an efficient state transfer mechanism from erasure coding.

We show our state transfer mechanism in Algorithm 3. We first introduce a new data structure called *succinct block* into HumBird-1. We define the *succinct block* for a block B as the tuple (h, h_c, t, cf) signed by the proposer of B, where h is the Merkle tree root corresponding to B, h_c is the Merkle tree root corresponding to the parent of B, t is the slot of B, and cf is the commit flag of B. The Merkle tree root corresponding to a block B is computed over the fragments of B that are generated by applying an $(f+1, n)$-erasure coding scheme to B.

Then we adapt the HumBird-1 protocol with succinct blocks. At the beginning of a slot t, an honest leader that proposes a block B will include the succinct block for B in its $\langle \text{PROPOSE} \rangle$ message before sending the message to all replicas. Once receiving the $\langle \text{PROPOSE} \rangle$ message, each honest replica first verifies the succinct block in the message. If so, each replica votes for B. Inside each vote, the succinct block for B is included. At the end of the slot, an honest replica will finalize B only if the replica has observed $f+1$ votes containing the succinct block for B, the commit flag of the succinct block is 1, and no other succinct block from slot t has been received by the replica.

To retrieve the original block of a succinct block, we can use erasure coding, in a similar way to many known protocols such as reliable broadcast [7, 17, 29]. The good-case latency of our protocol remains 2Δ when all replicas are honest or when there are only crash failures. When there are Byzantine failures, the latency is

Algorithm 3 State transfer mechanism. Code for p_i.

upon finalizing a block B corresponding to a succinct block b in the main protocol **do**
  if $\mathcal{T}$ contains the block B corresponding to b **then**
    let $\{d_j\}_{j=1,\dots n}$ be the fragments of an $(f+1, n)$-erasure coding scheme applied to B and h be the Merkle tree root computed over the fragments
    for $j = 1, \dots, n$ **do**
      send $m = \langle \text{SHARE}, h, d_j, \pi_j \rangle_i$ to replica p_j, where π_j is the j-th Merkle tree branch
upon receiving $\langle \text{SHARE}, h, d_i, \pi_i \rangle_*$ for the first time **do**
  if the block corresponding to h should be finalized
    and π_i is a valid Merkle branch for root h and leaf d_i
    and $d_i \notin D(h)$ **then**
      $D(h) \leftarrow D(h) \cup \{d_i\}$, where $D(h)$ is the set of all valid fragments with respect to h
      send message $m = \langle \text{SHARE}, h, d_i, \pi_i \rangle_i$ to all replicas
upon receiving $\langle \text{SHARE}, h, d_j, \pi_j \rangle_j$ from $p_j \neq p_i$ **do**
  if π_j is a valid Merkle branch for root h and leaf d_j
    and $d_j \notin D(h)$ **then**
      $D(h) \leftarrow D(h) \cup \{d_j\}$, where $D(h)$ is the set of all valid fragments with respect to h
      if $|D(h)| \geq f + 1$ **then**
        $B \leftarrow \boldsymbol{decode}(D(h))$
        $\mathcal{T} \leftarrow \mathcal{T} \cup \{B\}$

$2\Delta + 2\delta$, due to the state transfer mechanism. We show HumBird-1 with the state transfer mechanism and the proof of correctness in Appendix B.

4.4 Discussion

As mentioned in Sec. 3.2, we borrow some techniques from conventional BFT for building safe view changes to build HumBird-1. Our main insight is that building a reorg-resilient protocol can be much cheaper than building safe view changes. In the context of rotating-leader BFT, this insight indicates that the ForkChoice and the voting process can be made much cheaper. Namely, in our protocol, the leader can propose a new block and each replica can cast its vote based on their local physical clocks, rather than waiting for a sufficiently large number of messages. Reorg resilience can be achieved via forwarding the certificate during the voting process and the uncle blocks. As summarized in Table 1, our protocol enjoys much lower expected latency than existing works without sacrificing the good-case latency.

We do not claim too much novelty in building HumBird-1 with state transfer, as the techniques are quite similar to existing works [8, 22].

5 HumBird-2: Partially Synchronous Rotating Leader BFT

In this section, we present HumBird-2, a partially synchronous BFT variant of HumBird-1 under the $n \geq 3f + 1$ assumption. Without loss of generality, we assume $n = 3f + 1$. As shown in Table 2, compared to known partially synchronous rotating leader BFT protocols, HumBird-2 achieves the same good-case latency as the state-of-the-art protocols, and lower communication complexity than the state-of-the-art protocols.

Algorithm 4 The main protocol of HumBird-2. Code for p_i.

Global parameter: slot t
Local parameters: a block tree $\mathcal{T}$, a vote set $V(B)$ for each block B in $\mathcal{T}$
Initialize the following parameters:
  $\mathcal{T} \leftarrow \emptyset$, $t \leftarrow 1$, $\tau_1 \leftarrow 0$
▷ For the leader of slot t:
upon a slot t begins at time τ_t **do**
  let $(\text{head}, \text{candidate}, \text{cer}_c)$ be ForkChoice's output
  set the commit flag cf as 0 and uncle as $\perp$
  if slot t is an even slot **and** $slot(\text{cer}_c) < t - 1$ **then**
    let uncle be the block p_i proposes in slot $t - 1$
  if slot t is an even slot **or** $slot(\text{cer}_c) = t - 1$ **then**
    $cf \leftarrow 1$
  create $B = \langle \text{vals}, H(\text{candidate}), t, cf, \text{uncle} \rangle_i$
  send message $m = \langle \text{PROPOSE}, B, t, \text{cer}_c \rangle_i$ to all replicas
▷ For all replicas:
upon receiving the first valid $\langle \text{PROPOSE}, B, t, \text{cer} \rangle_j$ from the leader p_j of slot t before $\tau_t + \Delta$ **do**
  let $(\text{head}, \text{candidate}, \text{cer}_c)$ be ForkChoice's output
  if cer is a certificate for the parent of B **and** $slot(\text{cer}) \geq slot(\text{head})$
    and the uncle block of B is valid **then**
      forward the $\langle \text{PROPOSE} \rangle$ message to all replicas
      send vote $v = \langle \text{VOTE}, t, H(B) \rangle_i$ to all replicas
  else
    send cer_c to the leader p_j
upon receiving a valid $\langle \text{PROPOSE}, B, t, \text{cer} \rangle_j$ in slot t **do**
  $\mathcal{T} \leftarrow \mathcal{T} \cup \{B\}$
upon receiving a vote $v = \langle \text{VOTE}, t', h' \rangle_*$ **do**
  wait until the block B' corresponding to v is added to $\mathcal{T}$
  $V(B') \leftarrow V(B') \cup \{v\}$
upon observing a certificate for a block B of slot t before $\tau_t + 2\Delta$ **do**
  if the commit flag of B is 1 **then**
    send message $m = \langle \text{COMMIT}, t, H(B) \rangle_i$ to all replicas
upon receiving $2f + 1$ $\langle \text{COMMIT} \rangle$ messages for block B **do**
  if B has not been finalized **then**
    for each unfinalized ancestor of B **do**
      finalize the ancestor and its uncle block
    finalize B and the uncle block of B

We show the pseudocode of HumBird-2 in Algorithm 4. In this protocol, the workflow is quite similar to that in HumBird-1. We highlight the main differences in gray. The ForkChoice function of HumBird-2 is the same as that of HumBird-1.

The workflow. In the main protocol, we slightly change the workflow and introduce another $\langle \text{COMMIT} \rangle$ phase that is run in parallel with the protocol for each proposed block. Specifically, in a slot t, if each replica receives a certificate for a block B that is proposed in slot t and the commit flag of B is 1, it sends a $\langle \text{COMMIT} \rangle$ message to all replicas. Upon receiving $2f + 1$ $\langle \text{COMMIT} \rangle$ messages on block B, each replica finalizes B and all the blocks on the chain led by B. Here, the finalized blocks also include the corresponding uncle blocks.

The parallel $\langle \text{Commit} \rangle$ phase is conceptually similar to protocols such as PBFT [19], which involves three communication steps to agree on the order of each block and the agreement on different

blocks can be reached in parallel. Different from PBFT that has a high-communication and expensive view change process, HumBird-2 avoid expensive view changes due to the use of the LST model and uncle blocks.

Sketch of correctness. We begin with safety. Namely, after each leader proposes a block B, each replica sends a $\langle \text{VOTE} \rangle$ to all replicas. After receiving a certificate of $2f+1$ matching $\langle \text{VOTE} \rangle$ messages, each replica sends a $\langle \text{COMMIT} \rangle$ message to all replicas. Upon receiving $2f+1$ $\langle \text{COMMIT} \rangle$ messages for B, each replica finalizes B. According to quorum intersection, no honest replica can receive $2f+1$ $\langle \text{COMMIT} \rangle$ messages for any other block. Safety within each slot thus follows the proof of conventional BFT protocols. For safety across slots, the proof is also not too different. Namely, each replica will not vote for a block conflicting with its head block. If a block B is finalized in a slot, at least $f+1$ honest replicas will obtain head $= B$ from ForkChoice. Thus, any later block proposal conflicting with B will not be voted for by $2f+1$ replicas. Safety thus holds.

Liveness holds mainly because each replica sends its cer_c to the leader in the odd slot if the replica does not vote for the leader's block. Thus, after GST, the leader is guaranteed to receive the highest certified block, and its block proposed in the even slot can be finalized.

HumBird-2 achieves reorg resilience only after GST. Namely, since the block proposed in the even slot can be finalized, the block proposed in the odd slot can at least be included as an uncle block. Reorg resilience thus follows a similar argument as that in HumBird-1.

Latency and complexity. Compared to HumBird-1, we additionally introduce a $\langle \text{COMMIT} \rangle$ message. Meanwhile, each replica can vote upon receiving a block proposal. Thus, the good-case latency is 3δ. When there are $\lfloor \frac{n-1}{3} \rfloor$ faulty replicas, HumBird-2 has an expected latency of $3.11\Delta + 3\delta$. The full analysis of the expected latency is presented in Appendix F. For the communication complexity, we can also use the state transfer mechanism in Sec. 4.3 to optimize the communication. The complexity is $O(Ln + \kappa n^2 \log n)$.

We prove the correctness of HumBird-2 in Appendix D.

6 Implementation and Evaluation

Implementation and baseline details. We implement HumBird-1, HumBird-2, SyncRR [6], Sync HotStuff [2], and DispersedSimplex [50] in Golang. We use ECDSA as the digital signature, and use SHA256 as the underlying hash function.

The implementation details and the acronym we use in this section are summarized as follows.

RL-SHS: Sync HotStuff is not designed by default in the rotating leader mode. We implement a version where the leader is rotated every time a block is proposed and call it RL-SHS.

SRR-original: We implement the default SyncRR protocol. We call it SRR-original. Namely, SRR-original achieves $O(Ln^2 + \kappa n^3)$ communication.

SRR-ST: We implement a communication-efficient variant of SyncRR that achieves $O(Ln + \kappa n^3)$ communication. Specifically, SyncRR involves a communication step where each replica forwards its received block proposal to all replicas. We replace this step with one that each replica only forwards the signature on the hash of the proposal. The original proposal is obtained later via the state transfer mechanism mentioned in Sec. 4.3. We call this new version SRR-ST.

Table 3: The step durations of HB1-Asym in each slot.

Network setting	Duration of $\langle \text{PROPOSE} \rangle$ step	Duration of $\langle \text{VOTE} \rangle$ step
$n = 5, \Delta = 125$ ms	240 ms	10 ms
$n = 21, \Delta = 180$ ms	340 ms	20 ms
$n = 41, \Delta = 180$ ms	330 ms	30 ms
$n = 61, \Delta = 250$ ms	470 ms	30 ms

HB1-Asym and HB1-Sym: In our evaluation of HumBird-1, we implement asymmetric durations for the $\langle \text{PROPOSE} \rangle$ step and $\langle \text{VOTE} \rangle$ step. Specifically, HB1-Asym has a longer duration for the $\langle \text{PROPOSE} \rangle$ step and a shorter duration for the $\langle \text{VOTE} \rangle$ step. We use HB1-Sym to denote HumBird-1 with the same durations. The concrete durations of HB1-Asym are listed in Table 3. Unless otherwise mentioned, we use the symmetric mode when we use HumBird-1 in this section, where both the $\langle \text{PROPOSE} \rangle$ step and $\langle \text{VOTE} \rangle$ step have a duration of Δ.

HumBird-2: We implement our partially synchronous protocol HumBird-2 as in Algorithm 4.

DispersedSimplex: We implement a communication-efficient version of the Simplex protocol that achieves $O(Ln + \kappa n^3)$ communication, which is described in [50].

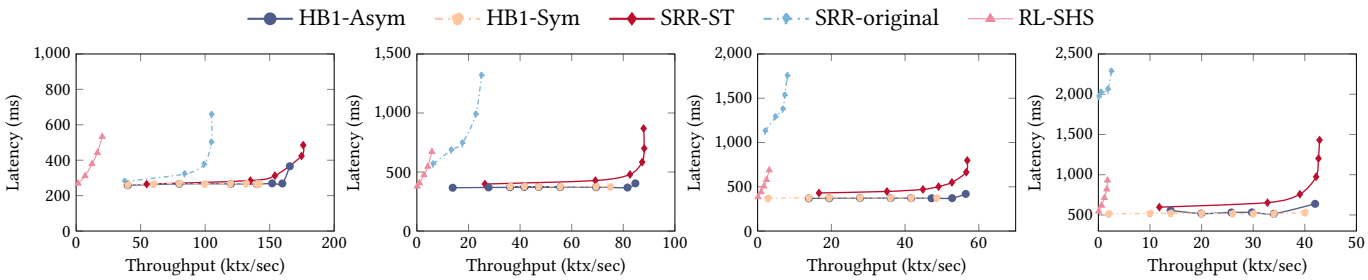
Evaluation setup. We evaluate the performance of the protocols on Amazon EC2 using up to 61 virtual machines (VMs). We use *m5.xlarge* instances for our evaluation. The m5.xlarge instance has four virtual CPUs and 16GB memory. To assess the protocols' performance in a synchronous network, we use a LAN setting, where all instances are deployed in the us-west-2 region.

We conduct experiments under different network sizes, number of failures, and batch sizes. We use n to denote the network size and f to denote the maximum number of Byzantine replicas in the network. For HumBird-1, SyncRR, and RL-SHS, we use $n = 2f + 1$ replicas. For HumBird-2 and DispersedSimplex, we use $n = 3f + 1$ replicas. We use b to denote the batch size, where each block has b transactions. The default transaction size is 128 bytes. In each experiment, we run each protocol until at least 100 blocks are finalized and report the average results.

6.1 Performance of the Synchronous Protocols in Failure-Free Cases

Good-case latency. We evaluate the good-case latency of HumBird-1, SRR-original, SRR-ST, and RL-SHS for $n = 21$ and $n = 41$. We set Δ as 180 *ms* and the batch size b as 5000. As shown in Figure 4, our HumBird-1 protocol consistently achieves a lower latency than all other protocols in the failure-free cases. When $n = 41$, HumBird-1 achieves 45.5% lower latency compared to RL-SHS, 72.7% lower latency compared to SRR-original, and 20.2% lower latency compared to SRR-ST. This matches our results in Table 1.

Meanwhile, the latency of HumBird-1 barely increases as n grows. In contrast, the latency of SRR-original and RL-SHS is 84.8% and 25.9% higher when n grows from 21 to 41, respectively. The nice



(a) $n = 5$. **(b)** $n = 21$. **(c)** $n = 41$. **(d)** $n = 61$.

Figure 3: Latency vs. throughput of HumBird-1, SyncRR, and RL-SHS under the failure-free scenario.

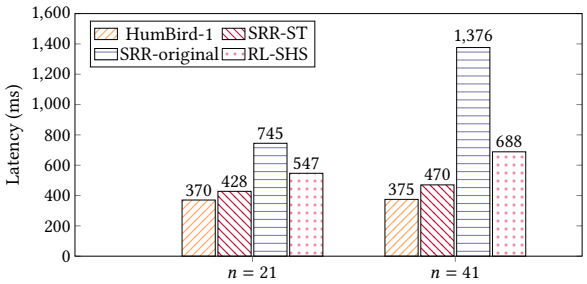


Figure 4: The good-case latency of HumBird-1, SyncRR, and RL-SHS under the failure-free scenario.

latency advantage is mainly because HumBird-1 has much lower communication complexity than SRR-original and RL-SHS.

Latency vs. throughput. We assess the latency vs. throughput of HB1-Asym, HB1-Sym, SRR-ST, SRR-original, and RL-SHS for $n = 5, 21, 41, 61$. Specifically, each protocol is evaluated across a range of batch sizes. For each batch size, we measure latency as the average time to finalize a block and throughput as the average number of finalized transactions per second. In our experiments, we increase the batch size until the system reaches the peak throughput. We set Δ as 125 *ms* for $n = 5$, set Δ as 180 *ms* for $n = 21$ and $n = 41$, and set Δ as 250 *ms* for $n = 61$. Note that it is *normal* to use different Δ for different n even for synchronous BFT [11, 12, 63].

We show the latency vs. throughput of the protocols in Figure 3. Our results demonstrate that our HumBird-1 protocol has a similar peak throughput to SRR-ST, while achieving much higher throughput than SRR-original and RL-SHS. For instance, when $n = 61$, the peak throughput of HB1-Asym is 16.7x and 23.8x that of SRR-original and RL-SHS, respectively. Additionally, HB1-Asym achieves a higher peak throughput than HB1-Sym. This is because HB1-Asym has a longer duration of the ⟨PROPOSE⟩ step in each slot. Thus, leaders in HB1-Asym can propose larger blocks than those in HB1-Sym without triggering a timeout.

6.2 Performance of the Synchronous Protocols under Failures

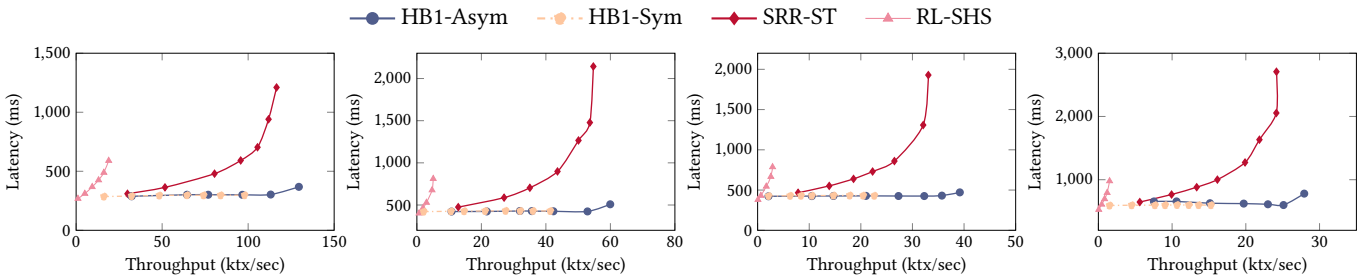
We assess the performance of HumBird-1, SyncRR, and RL-SHS under multiple failure scenarios. In particular, we evaluate the protocols in both a moderate-failure case (i.e., $\lfloor \frac{n}{4} \rfloor$ failures) and a high-failure case (i.e., $\lfloor \frac{n-1}{2} \rfloor$ failures). For each number of failures, we further test the impact of two different failure scenarios. In the first scenario, faulty replicas are selected randomly. In the second scenario, faulty replicas are limited to the ones with odd IDs. Within all the failure scenarios, we let faulty replicas remain crashed throughout the execution. Evaluation under crash failures follows the convention of the literature [13, 23, 24].

Latency vs. throughput. We assess the latency vs. throughput of HB1-Asym, HB1-Sym, SRR-ST, and RL-SHS under four failure scenarios. Within each failure scenario, we evaluate the performance of the protocols for $n = 5, 21, 41, 61$. For each n, we use the same value of Δ as in the failure-free scenario. We report the results under the failure scenarios where faulty replicas are limited to the ones with odd IDs in Figure 5 and Figure 6. Additionally, we show the results under the failure scenarios where the faulty replicas are selected randomly in Appendix E.

Our results show that HumBird-1 consistently achieves the highest peak throughput among all protocols. For example, when $n = 61$, the number of faulty replicas is $\lfloor \frac{n-1}{2} \rfloor$, and only replicas with odd-numbered IDs can become faulty, the peak throughput of HB1-Asym is about 1.7x that of SRR-ST and 9.7x that of RL-SHS. This is expected, as HumBird-1 does not involve an explicit view change process. In addition, all the protocols suffer performance degradation as the number of faulty replicas increase, but to different degrees. The performance degradation of HumBird-1 is consistently lower than all other protocols. For instance, when $n = 61$ and the number of faulty replicas grows from 0 to $\lfloor \frac{n-1}{2} \rfloor$, the peak throughput of HB1-Asym decreases by at most 59.4%, while the throughput of SRR-ST drops by 78.2%.

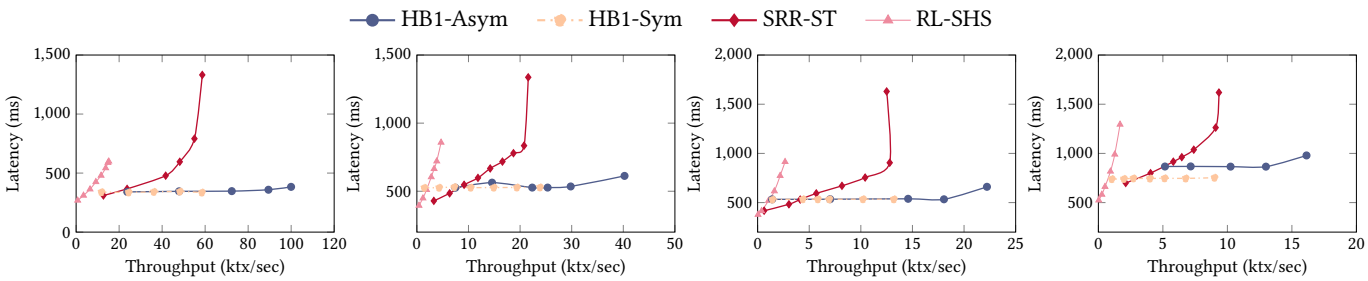
Average latency. We evaluate the average latency of HumBird-1, SRR-ST, and RL-SHS under $\lfloor \frac{n-1}{2} \rfloor$ failures with $n = 21$ and $n = 41$. We employ random leader election in all protocols and measure the finalization latency of transactions submitted before the fifth leader election in each experiment. The experiment is repeated at least 60 times and we report the average result. We set $b = 5000$ and $\Delta = 180$ ms.

The average latency of the protocols is shown in Figure 7. Our results demonstrate that our HumBird-1 protocol consistently achieves a lower average latency than all other protocols. In particular, when $n = 41$, HumBird-1 achieves 41.7% lower latency compared to RL-SHS and 31.9% lower latency compared to SRR-ST. This matches the theoretical results in Table 1. Additionally, the average latency



(a) $n = 5$. **(b)** $n = 21$. **(c)** $n = 41$. **(d)** $n = 61$.

Figure 5: Latency vs. throughput of HumBird-1, SyncRR, and RL-SHS under the failure scenario where the number of faulty replicas is $\lfloor \frac{n}{4} \rfloor$ and only replicas with odd-numbered IDs can become faulty.



(a) $n = 5$. **(b)** $n = 21$. **(c)** $n = 41$. **(d)** $n = 61$.

Figure 6: Latency vs. throughput of HumBird-1, SRR-ST, and RL-SHS under the failure scenario where the number of faulty replicas is $\lfloor \frac{n-1}{2} \rfloor$ and only replicas with odd-numbered IDs can become faulty.

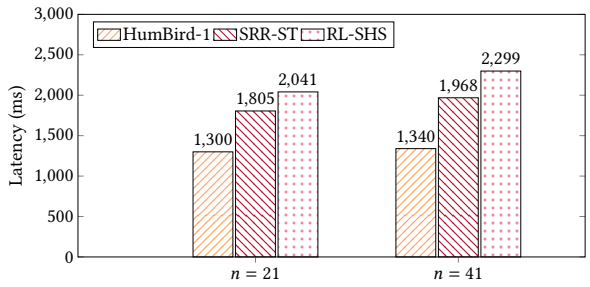


Figure 7: The average latency of HumBird-1, SRR-ST, and RL-SHS under failures.

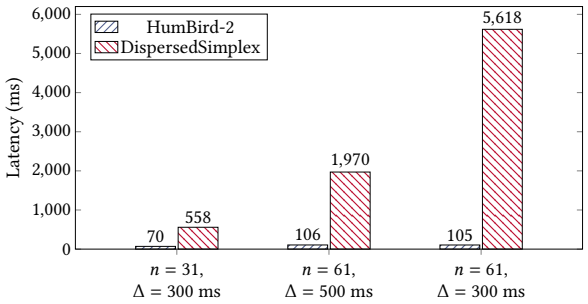


Figure 8: The good-case latency of HumBird-2 and DispersedSimplex under the failure-free case.

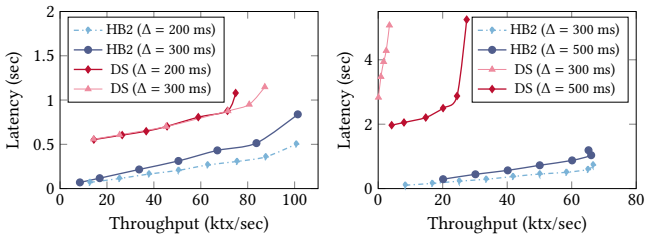
of HumBird-1 barely increases as n grows. In contrast, the latency of SRR-ST and RL-SHS is 9.0% and 12.6% higher when n grows from 21 to 41, respectively.

6.3 Performance of the Partially Synchronous Protocols in Failure-Free Cases

Good-case latency. We evaluate the good-case latency of HumBird-2 and DispersedSimplex for $n = 31$ and $n = 61$. We set the batch size b as 5000. As shown in Figure 8, our HumBird-2 protocol achieves a much lower good-case latency than DispersedSimplex in failure-free cases. In particular, HumBird-2 achieves 87.5% lower latency than DispersedSimplex when $n = 31$ and $\Delta = 300$ ms, and achieves 94.6% lower latency than DispersedSimplex when $n = 61$ and $\Delta = 500$ ms. The results are expected, as HumBird-2 also has lower communication complexity.

Additionally, our results show that HumBird-2 remains efficient when Δ decreases, while DispersedSimplex suffers significant performance degradation. In particular, when $n = 61$, the good-case latency of HumBird-2 remains stable as Δ drops from 500 ms to 300 ms, while the latency of DispersedSimplex increases by 285%. This is because with a smaller Δ, many blocks in DispersedSimplex cannot be finalized before their view timers expire. In contrast, blocks in HumBird-2 can still be finalized.

Latency vs. throughput. We assess the peak throughput of HumBird-2 and DispersedSimplex for $n = 31$ and $n = 61$. As shown in Figure 9, our HumBird-2 protocol has higher peak throughput than



(a) $n = 31$. **(b)** $n = 61$.

Figure 9: Latency vs. throughput of HumBird-2 and DispersedSimplex under the failure-free case. We use HB2 to denote HumBird-2 and DS to denote DispersedSimplex.

DispersedSimplex. For instance, when $n = 61$ and $\Delta = 500$ ms, the peak throughput of HumBird-2 is 2.41x that of DispersedSimplex. Additionally, the peak throughput of HumBird-2 remains stable as Δ decreases, while that of DispersedSimplex declines significantly. In particular, when $n = 61$ and Δ decreases from 500 ms to 300 ms, the peak throughput of DispersedSimplex drops by 87.3%. This is because when Δ decreases to 300 ms, votes from honest replicas cannot be collected before the view timer expires. Therefore, blocks cannot be finalized easily. In comparison, our uncle block design can well mitigate such an issue.

7 Additional Related Work

Synchronous leader-based BFT. Conventional synchronous BFT protocols [2, 4, 5, 21, 51] are leader-based. Most of them [2, 4, 51] follow a stable-leader paradigm to optimize the performance of the normal case. The state-of-the-art solutions achieve a latency close to 1Δ [2, 4]. However, the performance of stable-leader BFT degrades significantly under a slow or malicious leader.

Many rotating leader protocols focus on improving the good-case latency. Besides SyncRR [6], OptSync [51] achieves a good-case latency of $3\Delta + \delta$ when $f < n/2$, and a good-case latency of 3δ when $f < n/4$. Both SyncRR and OptSync have much higher expected latency than their good-case latency. When $f < n/2$, SyncRR achieves an expected latency of $10.5\Delta + 3\delta$ and OptSync achieves an expected latency of $11\Delta + 4\delta$. In contrast, our HumBird-1 protocol achieves a good-case latency of 2Δ and an expected latency of 7.5Δ when $f < n/2$.

Some works also try to optimize the duration of the view change process. Shi's protocol [49] can rotate leaders every 1Δ, regardless of whether the leader is honest or faulty. Streamlet [21] rotates leaders every 2Δ. However, Shi's protocol has a good-case latency of $O(n\Delta)$ and Streamlet has a good-case latency of 12Δ, both of which are much higher than HumBird-1. Additionally, Shi's protocol and Streamlet are not reorg-resilient. In contrast, our HumBird-1 protocol achieves reorg resilience and avoids an explicit view change process.

Another line of work focuses on improving the communication complexity. RandPiper [12] introduces a BFT SMR protocol with a communication complexity of $O(Ln + \kappa n^2 \log n)$. However, RandPiper has an expected latency of $17\Delta + 3\delta$, which is much higher than HumBird-1.

A concurrent work [3] proposes four synchronous rotating-leader BFT protocols. In the $n = 2f + 1$ setting, these protocols can achieve a good-case latency of $O(\delta)$ when all replicas are honest. However, when there are faulty replicas, the protocols either have a higher good-case latency than HumBird-1, or may experience a longer view change process under faulty leaders.

Partially synchronous rotating-leader BFT. Many studies [18, 20, 30, 35, 50, 57] propose rotating-leader BFT protocols within partially synchronous networks. Recent works [18, 20, 30, 50] try to reduce the good-case latency. ICC0 and Commit Moonshot achieve the optimal good-case latency of 3δ in the partially synchronous setting for $f < n/3$ [5]. However, these protocols are not reorg-resilient even when the network is synchronous. The only known reorg-resilient partially synchronous BFT protocols are Simplex [20] and DispersedSimplex [50]. Compared to Simplex and DispersedSimplex, our HumBird-2 protocol achieves the same good-case latency and lower communication complexity.

Reorg-resilient protocols. The notion of reorg resilience is due to Goldfish [26]. A synchronous protocol is presented. A follow-up work RLMD-GHOST [28] later provides a solution in the partially synchronous model. AA [61] provides an efficient solution for Ethereum PoS that achieves reorg resilience when the network is synchronous. In our paper, we follow the definition of reorg resilience in AA to build our HumBird-1 and HumBird-2 protocol. For our partially synchronous protocol HumBird-2, reorg resilience is achieved also when the network is synchronous.

DAG-based BFT. Directed acyclic graph (DAG) based BFT are known to use the multi-proposer diagram to improve the throughput and many DAG protocols are studied in the partially synchronous network [9, 37, 52–55]. Many recent DAG-based protocols seek to reduce the latency [9, 37, 52]. The lowest latency known so far is due to Sailfish [52], which finalizes leader blocks with a latency of 3δ and finalizes non-leader blocks with a latency of 5δ.

8 Conclusion

We present HumBird, a family of rotating leader BFT protocols. We show that by employing the leader-speaks-twice model and defining an uncle block field, we can build rotating leader BFT protocols with reorg resilience. As a result, our protocols are simple and achieve low latency. We present a synchronous BFT HumBird-1 and a partially synchronous BFT HumBird-2. Our experimental results show that compared to the state-of-the-art rotating leader BFT protocols, HumBird-1 and HumBird-2 achieve up to 32% lower latency and 70% higher throughput, and 95% lower latency and 140% higher throughput, respectively.

References

[1] Ittai Abraham, Srinivas Devadas, Danny Dolev, Kartik Nayak, and Ling Ren. 2019. Synchronous Byzantine Agreement with Expected O(1) Rounds, Expected Communication, and Optimal Resilience. In *FC*. 320–334.

[2] Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Maofan Yin. 2020. Sync HotStuff: Simple and Practical Synchronous State Machine Replication. In *S&P*. 106–118.

[3] Ittai Abraham, Kartik Nayak, Ling Ren, and Ertem Nusret Tas. 2026. Synchronous BFT with Provability and Fast Path for the Age of Blockchains. Cryptology ePrint Archive, Paper 2026/639.

[4] Ittai Abraham, Kartik Nayak, Ling Ren, and Zhuolun Xiang. 2020. Byzantine Agreement, Broadcast and State Machine Replication with Near-optimal Good-Case Latency. *arXiv preprint arXiv:2003.13155* (2020).

[5] Ittai Abraham, Kartik Nayak, Ling Ren, and Zhuolun Xiang. 2021. Good-case Latency of Byzantine Broadcast: A Complete Categorization. In *PODC*. 331–341.
[6] Ittai Abraham, Kartik Nayak, and Nibesh Shrestha. 2022. Optimal Good-Case Latency for Rotating Leader Synchronous BFT. In *OPODIS*, Vol. 217. 27:1–27:19.
[7] Nicolas Alhaddad, Sourav Das, Sisi Duan, Ling Ren, Mayank Varia, Zhuolun Xiang, and Haibin Zhang. 2022. Balanced Byzantine Reliable Broadcast with Near-Optimal Communication and Improved Computation. In *PODC*. 399–417.
[8] Nicolas Alhaddad, Sisi Duan, Mayank Varia, and Haibin Zhang. 2021. Succinct Erasure Coding Proof Systems. Cryptology ePrint Archive, Paper 2021/1500.
[9] Kushal Babel, Andrey Chursin, George Danezis, Anastasios Kichidis, Lefteris Kokoris-Kogias, Arun Koshy, Alberto Sonnino, and Mingwei Tian. 2025. Mysticeti: Reaching the Latency Limits with Uncertified DAGs. In *NDSS*.
[10] Adithya Bhat, Akhil Bandarupalli, Saurabh Bagchi, Aniket Kate, and Michael K. Reiter. 2023. The Unique Chain Rule and Its Applications. In *FC*. 38–55.
[11] Adithya Bhat, Nibesh Shrestha, Aniket Kate, and Kartik Nayak. 2023. Optrand: Optimistically responsive reconfigurable distributed randomness. In *NDSS*.
[12] Adithya Bhat, Nibesh Shrestha, Zhongtang Luo, Aniket Kate, and Kartik Nayak. 2021. RandPiper: Reconfiguration-Friendly Random Beacons with Quadratic Communication. In *CCS*. 3502–3524.
[13] Erica Blum, Jonathan Katz, Julian Loss, Kartik Nayak, and Simon Ochsenreither. 2023. Abraxas: Throughput-Efficient Hybrid Asynchronous Consensus. In *CCS*. 519–533.
[14] Vitalik Buterin and Virgil Griffith. 2017. Casper the friendly finality gadget. *arXiv preprint arXiv:1710.09437* (2017).
[15] Vitalik Buterin, Diego Hernandez, Thor Kamphefner, Khiem Pham, Zhi Qiao, Danny Ryan, Juhyeok Sin, Ying Wang, and Yan X Zhang. 2020. Combining GHOST and casper. *arXiv preprint arXiv:2003.03052* (2020).
[16] Christian Cachin, Klaus Kursawe, and Victor Shoup. 2005. Random Oracles in Constantinople: Practical Asynchronous Byzantine Agreement Using Cryptography. *Journal of Cryptology* 18, 3 (2005).
[17] Christian Cachin and Stefano Tessaro. 2005. Asynchronous verifiable information dispersal. In *SRDS*. 191–201.
[18] Jan Camenisch, Manu Drijvers, Timo Hanke, Yvonne-Anne Pignolet, Victor Shoup, and Dominic Williams. 2022. Internet Computer Consensus. In *PODC*. 81–91.
[19] Miguel Castro and Barbara Liskov. 1999. Practical byzantine fault tolerance. In *OSDI*. 173–186.
[20] Benjamin Y Chan and Rafael Pass. 2023. Simplex consensus: A simple and fast consensus protocol. In *TCC*. 452–479.
[21] Benjamin Y. Chan and Elaine Shi. 2020. Streamlet: Textbook Streamlined Blockchains. In *AFT*. 1–11.
[22] Daniel Collins, Sisi Duan, Julian Loss, Charalampos Papamanthou, Giorgos Tsimos, and Haochen Wang. 2025. Towards Optimal Parallel Broadcast Under a Dishonest Majority. In *FC*. 345–363.
[23] Xiaohai Dai, Chaozheng Ding, Hai Jin, Julian Loss, and Ling Ren. 2024. Ipotane: Balancing the Good and Bad Cases of Asynchronous BFT. In *NDSS*.
[24] Xiaohai Dai, Bolin Zhang, Hai Jin, and Ling Ren. 2023. ParBFT: Faster Asynchronous BFT Consensus with a Parallel Optimistic Path. In *CCS*. 504–518.
[25] Philip Daian, Steven Goldfeder, Tyler Kell, Yunqi Li, Xueyuan Zhao, Iddo Bentov, Lorenz Breidenbach, and Ari Juels. 2020. Flash boys 2.0: Frontrunning in decentralized exchanges, miner extractable value, and consensus instability. In *S&P*. 910–927.
[26] Francesco D'Amato, Joachim Neu, Ertem Nusret Tas, and David Tse. 2024. Goldfish: No More Attacks on Ethereum?!. In *FC*. 3–23.
[27] Francesco D'Amato, Roberto Saltini, Thanh-Hai Tran, and Luca Zanolini. 2025. TOB-SVD: Total-Order Broadcast with Single-Vote Decisions in the Sleepy Model. In *ICDCS*. 1033–1043.
[28] Francesco D'Amato and Luca Zanolini. 2024. Recent Latest Message Driven GHOST: Balancing Dynamic Availability With Asynchrony Resilience. In *CSF*. 127–142.
[29] Sourav Das, Zhuolun Xiang, and Ling Ren. 2021. Asynchronous Data Dissemination and Its Applications. In *CCS*. 2705 – 2721.
[30] Isaac Doidge, Raghavendra Ramesh, Nibesh Shrestha, and Joshua Tobkin. 2024. Moonshot: Optimizing Block Period and Commit Latency in Chain-Based Rotating Leader BFT. In *DSN*. 470–482.
[31] Sisi Duan, Michael K Reiter, and Haibin Zhang. 2017. Secure causal atomic broadcast, revisited. In *DSN*. 61–72.
[32] Sisi Duan and Haibin Zhang. 2022. Foundations of Dynamic BFT. In *S&P*. 1317–1334.
[33] Sisi Duan, Haibin Zhang, Xiao Sui, Baohan Huang, Changchun Mu, Gang Di, and Xiaoyun Wang. 2024. Dashing and Star: Byzantine Fault Tolerance with Weak Certificates. In *EuroSys*. 250–264.
[34] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. 1988. Consensus in the presence of partial synchrony. *Journal of ACM* 35, 2 (1988), 288–323.
[35] Francesco D'Amato and Luca Zanolini. 2023. A Simple Single Slot Finality Protocol for Ethereum. In *ESORICS Workshops*. 376–393.
[36] Neil Giridharan, Florian Suri-Payer, Matthew Ding, Heidi Howard, Ittai Abraham, and Natacha Crooks. 2023. BeeGees: Stayin' Alive in Chained BFT. In *PODC*. 233–243.
[37] Idit Keidar, Oded Naor, Ouri Poupko, and Ehud Shapiro. 2023. Cordial Miners: Fast and Efficient Consensus for Every Eventuality. In *DISC*. 26:1–26:22.
[38] Rujia Li, Jingyuan Ding, Qin Wang, Keting Jia, Haibin Zhang, and Sisi Duan. 2025. Does finality gadget finalize your block? a case study of binance consensus. In *USENIX Security*. 4109–4125.
[39] Rujia Li, Xuanwei Hu, Qin Wang, Sisi Duan, and Qi Wang. 2026. Transaction Fairness in Blockchains, Revisited. *IEEE Transactions on Dependable and Secure Computing* 23, 1 (2026), 752–765.
[40] Rujia Li, Qin Wang, Haochen Wang, Xueqian Lu, and Sisi Duan. 2026. Greedy Attack: Breaking Finality against VeChain Proof-of-Authority Consensus Protocol. In *Proceedings of the ACM Web Conference 2026*. 3218–3228.
[41] Rujia Li, Mingfei Zhang, Xueqian Lu, Wenbo Xu, Ying Yan, and Sisi Duan. 2026. BunnyFinder: Finding Incentive Flaws for Ethereum Consensus. In *NDSS*.
[42] Dahlia Malkhi, Atsuki Momose, and Ling Ren. 2023. Towards Practical Sleepy BFT. In *CCS*. 490–503.
[43] Silvio Micali, Michael Rabin, and Salil Vadhan. 1999. Verifiable random functions. In *FOCS*. 120–130.
[44] Atsuki Momose and Ling Ren. 2022. Constant Latency in Sleepy Consensus. In *CCS*. 2295–2308.
[45] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system.
[46] Joachim Neu, Ertem Nusret Tas, and David Tse. 2021. Ebb-and-flow protocols: A resolution of the availability-finality dilemma. In *S&P*. 446–465.
[47] Michael Neuder, Daniel J. Moroz, Rithvik Rao, and David C. Parkes. 2020. Defending Against Malicious Reorgs in Tezos Proof-of-Stake. In *AFT*. 46–58.
[48] Michael Neuder, Daniel J. Moroz, Rithvik Rao, and David C. Parkes. 2021. Low-cost attacks on Ethereum 2.0 by sub-1/3 stakeholders. arXiv:2102.02247 [cs.CR]
[49] Elaine Shi. 2019. Analysis of deterministic longest-chain protocols. In *CSF*. 122–135.
[50] Victor Shoup. 2024. Sing a Song of Simplex. In *DISC*. 37:1–37:22.
[51] Nibesh Shrestha, Ittai Abraham, Ling Ren, and Kartik Nayak. 2020. On the Optimality of Optimistic Responsiveness. In *CCS*. 839–857.
[52] Nibesh Shrestha, Rohan Shrothrium, Aniket Kate, and Kartik Nayak. 2025. Sailfish: Towards Improving the Latency of DAG-Based BFT. In *S&P*. 1928–1946.
[53] Alexander Spiegelman, Balaji Arun, Rati Gelashvili, and Zekun Li. 2025. Shoal: Improving DAG-BFT Latency and Robustness. In *FC*. 92–109.
[54] Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. 2022. Bullshark: DAG BFT Protocols Made Practical. In *CCS*. 2705–2718.
[55] Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. 2022. Bullshark: The Partially Synchronous Version. arXiv:2209.05633 [cs.DC]
[56] Chenxu Wang, Sisi Duan, Minghui Xu, Feng Li, and Xiuzhen Cheng. 2026. Consensus in the Known Participation Model with Byzantine Faults and Sleepy Replicas. In *NDSS*.
[57] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. 2019. HotStuff: BFT Consensus with Linearity and Responsiveness. In *PODC*. 347–356.
[58] Jingfan Yu, Sisi Duan, and Zhixuan Fang. 2025. Efficient Fair Ordering Protocol with 2-hop Receiver Fairness. In *SRDS*. 195–206.
[59] Mingfei Zhang, Rujia Li, and Sisi Duan. 2024. Max Attestation Matters: Making Honest Parties Lose Their Incentives in Ethereum PoS. In *USENIX Security*. 6255–6272.
[60] Mingfei Zhang, Rujia Li, Xinyu Lei, and Sisi Duan. 2026. Risk-free Selfish Mining in Hybrid Predictability Model. A Case Study on Polkadot's NPoS. In *Proceedings of the ACM Web Conference 2026*. 3299–3310.
[61] Mingfei Zhang, Rujia Li, Xueqian Lu, and Sisi Duan. 2025. Available Attestation: Towards a Reorg-Resilient Solution for Ethereum Proof-of-Stake. In *USENIX Security*. 1263–1280.
[62] Mingfei Zhang, Rujia Li, Xueqian Lu, and Sisi Duan. 2026. A Liveness Attack to Ethereum PoS with No Additional Cost.. In *S&P*.
[63] Xufeng Zhang, Baohan Huang, Sisi Duan, and Haibin Zhang. 2025. Randomized vs. Deterministic? Practical Randomized Synchronous BFT in Expected Constant Time. In *SRDS*. 24–35.

A ForkChoice vs. Safe View Change

We use Sync HotStuff [2] as an example to discuss the relationship between ForkChoice and safe view change. Sync HotStuff is a synchronous BFT protocol assuming $n \geq 2f+1$. The normal case of the protocol for some view v is summarized as follows, where the crucial workflow why reorganization will not occur is highlighted in gray.

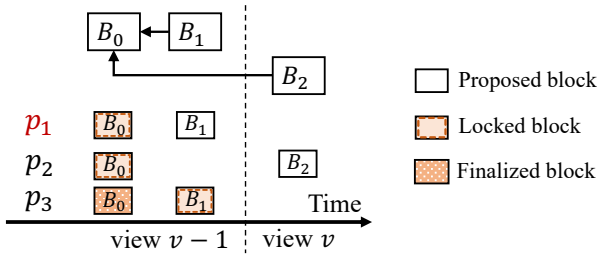


Figure 10: An example of Sync HotStuff. B_0 is the highest certified block that is known by p_1, p_2, and p_3 when they enter view $v-1$. In view $v-1$, the Byzantine leader p_1 proposes a block B_1 extending B_0. Only one honest replica p_3 receives a certificate for B_1 before the end of view $v-1$. When they enter view v, p_1 and p_2 are locked on B_0 and p_3 is locked on B_1. If the next leader p_2 is not aware of B_1, it will propose a block B_2 extending B_0, which will not be voted for by p_3. The view change process is thus mainly designed for an honest leader to obtain the *hidden lock*.

Propose. As the leader of view v, wait until it receives a valid certificate for its highest block B , create a block proposal B' that extends B, and send it to all replicas. Here, a certificate denotes $f+1$ matching votes on B.

Vote. Upon receiving a valid block B' that is not conflicting with its locked block, if no equivocation is detected, forward the proposal to all replicas, and send a vote on B' to all replicas. Wait for 2Δ time. Here, each replica is locked on some block B'' where B'' is its highest block with a valid certificate in view $v-1$. The highest block with a valid certificate is also called the *highest certified block*.

Finalization. If the 2Δ timer expires, finalize B' and all the blocks on the chain led by B'.

View change timer. Start a view change timer at the beginning of view v. If the leader is suspected to be faulty or fewer than some expected number of votes are created, start the view change process.

It can be seen from the workflow that the leader does not have to propose a block *on time*, e.g., at the beginning of some slot. Instead, during the view change from view $v-1$ to v, each replica will send its highest certified block with a valid certificate to the leader of view v. The leader selects one block B and sends B to the replicas, and the replicas cast a vote on the block. Only after a certificate is formed for B, the leader proceeds to view v. Such a process mainly addresses the issue where the leader of view v does not necessarily know the *highest* block locked by some honest replica, as shown in Figure 10. Via the votes during view changes, the new leader can safely extend a block, and its block proposal will be voted for by *all* honest replicas.

The drawback is that the view change process can be very lengthy, as mentioned in the introduction. In Sync HotStuff, each replica sets up a view change timer of 6Δ, and each view change takes $3\Delta+3\delta$ time. It becomes very expensive to use Sync HotStuff in the rotating leader mode in this way, according to the RL-SHS line in Table 1. Follow-up works such as SyncRR [6] improve the latency of the view change to $2\Delta+\delta$. However, each replica still sets up a view change timer of 7Δ, making the slot in which a faulty leader is selected very long.

B HumBird-1 with state transfer mechanism

We show HumBird-1 with the state transfer mechanism in Algorithm 5. We highlight the differences between HumBird-1 with and without the state transfer mechanism in gray. The main difference is that in HumBird-1 with the state transfer mechanism, the block in each vote is replaced with its corresponding succinct block. Additionlly, at the end of each slot, an honest replica will only finalize a block if it has observed $f+1$ votes containing the corresponding succinct block, the commit flag of the succinct block is 1, and no other succinct block from the same slot has been received by the replica.

Sketch of correctness. The safety of the modified HumBird-1 protocol remains equivalent to the original HumBird-1. The changes do not compromise the safety of the original protocol as it preserves the equivocation detection mechanism. In particular, if two honest replicas vote for two equivocating blocks, the succinct blocks in their votes will also conflict, making the equivocation detectable by all replicas. The liveness of the modified HumBird-1 relies on the termination of the state transfer mechanism, as every honest replica can only finalize a block after obtaining the full block from the state transfer mechanism. Our state transfer mechanism ensures termination when the proposer of a slot is honest and all honest replicas finalize some block at the end of the slot (According to Lemma 1).

Lemma 1. *If all honest replicas finalizes a block B at the end of a slot, each honest replica can reconstruct the block B, all ancestors of B, and all their uncle blocks eventually.*

Proof. Suppose all honest replicas finalizes a block B at the end of slot t. According to the protocol, $f+1$ replicas must have voted for the succinct block for B in slot t. Therefore, there must be an honest replica p_i among them, and p_i must have received the full block before voting for B. p_i will encode B to n fragments by applying a $(f+1, n)$-erasure coding scheme, compute a Merkle tree over the fragments, and disperse each fragment as well as the Merkle tree root and the branch to the corresponding replica. At least $f+1$ honest replicas will verify their fragments and forward the fragments to all replicas. According to the property of erasure coding, all honest replicas can reconstruct B when receiving $f+1$ valid fragments with respect to B. As B includes its uncle block, all honest replicas can also reconstruct the uncle block of B.

According to the protocol, there must have been a certificate for each ancestor of B (denoted as B_j). Therefore, there must be an honest replica p_j that has received the full block B_j before voting. Similarly, all honest replicas can reconstruct B_j and the uncle block of B_j.

□

C Correctness of HumBird-1

Lemma 2. *If an honest replica p_i finalizes a block B of slot t by the end of slot t, another honest replica p_j receives a certificate for a block B' such that $slot(B') = slot(B)$, then $B = B'$.*

Algorithm 5 HumBird-1 with state transfer. Code for proposer p_i.

Global parameter: slot t
Local parameters: a block tree $\mathcal{T}$, a succinct block set $\mathcal{S}$, a vote set $V(b)$ for each succinct block $b \in \mathcal{S}$

Initialize the following parameters:
    $\mathcal{T} \leftarrow \emptyset$, $\mathcal{S} \leftarrow \emptyset$, $t \leftarrow 1$, $\tau_1 \leftarrow 0$

▷ For the proposer of slot t:
upon a slot t begins at time τ_t **do**
    let (head, candidate, cer_c) be ForkChoice's output
    set the commit flag cf as 0 and uncle as $\perp$
    if slot t is an even slot **and** $slot(\text{cer}_c) < t-1$ **then**
        let uncle be the block p_i proposes in slot $t-1$
    if slot t is an even slot **or** $slot(\text{cer}_c) = t-1$ **then**
        $cf \leftarrow 1$
    let h_c be the Merkle tree root corresponding to candidate

    create block $B = \langle \text{vals}, h_c, t, cf, \text{uncle} \rangle_i$
    let $\{d_j\}_{j=1,\dots n}$ be the fragments of an $(f+1, n)$-erasure coding scheme applied to B and h be the Merkle tree root computed over the fragments
    let $b = \langle h, h_c, t, cf \rangle_i$ be the succinct block for B
    send message $m = \langle \text{PROPOSE}, B, t, \text{cer}_c, b \rangle_i$ to all replicas

▷ For all replicas:
upon receiving the first valid $\langle \text{PROPOSE}, B, t, \text{cer}, b \rangle_j$ from the proposer p_j of slot t before $\tau_t + \Delta$ **do**
    let $\{d_j\}_{j=1,\dots n}$ be the fragments of an $(f+1, n)$-erasure coding scheme applied to B and h be the Merkle tree root computed over the fragments
    if the Merkle tree root in b is not h **or** $slot(b) \neq t$ **or** the other parameters of b are inconsistent with those of B **then**
        abort
    $\mathcal{T} \leftarrow \mathcal{T} \cup \{B\}$, $\mathcal{S} \leftarrow \mathcal{S} \cup \{b\}$
    wait until $\tau_t + \Delta$
    let (head, candidate, cer_c) be ForkChoice's output
    if cer is a certificate for the parent of B **and** $slot(\text{cer}) \geq slot(\text{head})$
            and the uncle block of B is valid **then**
        send vote $v = \langle \text{VOTE}, t, b \rangle_i$ to all replicas
    else
        send cer_c to the proposer p_j
upon receiving $v = \langle \text{VOTE}, t', b' \rangle_*$ **do**
    if b' is valid **and** $slot(b') = t'$ **then**
        $\mathcal{S} \leftarrow \mathcal{S} \cup \{b'\}$, $V(b') \leftarrow V(b') \cup \{v\}$
upon a slot t ends at $\tau_t + 2\Delta$ **do**
    if $\mathcal{S}$ contains a succinct block b of slot t such that $|V(b)| \geq f+1$
            and no other succinct blocks of slot t are in $\mathcal{S}$
            and the commit flag of b is 1 **then**
        send $V(b)$ to all replicas
        if the block B corresponding to b is not finalized **then**
            execute the state transfer mechanism in Algorithm 3 until B, all B's ancestors, and all their uncle blocks are added to $\mathcal{T}$
            for each unfinalized ancestor of B **do**
                finalize the ancestor and its uncle block
            finalize B and its uncle block

Proof. We prove the correctness by contradiction. In HumBird-1, a certificate contains $f+1$ votes. The certificate for B' must contain $f+1$ votes for B' and at least one honest replica must have sent a $\langle \text{VOTE} \rangle$ message for B' before $\tau_t + \Delta$. p_i must have received the $\langle \text{VOTE} \rangle$ message for B' and added B' into $\mathcal{T}$ of p_i by the end of slot t. As $B' \neq B$, according to the protocol, p_i will not finalize B at the end of slot t, a contradiction. □

Corollary 1. *If an honest replica p_i finalizes a block B of slot t at the end of slot t, another honest replica p_j finalizes a block B' of slot t at the end of slot t, $B = B'$.*

Proof. According to the protocol, p_j must have received a certificate for B' before finalizing B'. Since $slot(B') = slot(B)$, by Lemma 2, we have $B = B'$. □

Lemma 3. *If an honest replica finalizes a block B of slot t by the end of slot t, each honest replica will obtain a head from ForkChoice where $slot(head) \geq t$ after $\tau_{t+1} + \Delta$.*

Proof. If an honest replica finalizes block B by the end of slot t, the replica must have collected a certificate for B and the commit flag of B must be 1. The replica will forward the certificate for B and all honest replicas will receive the certificate before $\tau_{t+1} + \Delta$. Since B is finalized at the end of slot t, according to Lemma 2, no certificate for any other block of slot t will be received by any honest replica. Therefore, according to the ForkChoice function, all honest replicas must set their head as B when $\tau_{t+1} + \Delta$.

After $\tau_{t+1} + \Delta$, honest replicas will not set head as any block B' where $slot(B') < t$. Therefore, each honest replica will obtain a head from ForkChoice where $slot(\text{head}) \geq t$ after $\tau_{t+1} + \Delta$. □

Lemma 4. *If an honest replica p_1 finalizes a block B_1 of slot t_1 at the end of slot t_1, another honest replica p_2 finalizes a block B_2 of slot $t_2 > t_1$ at the end of slot t_2, B_2 must extend B_1.*

Proof. We prove the correctness via contradiction. As p_2 finalizes B_2 at the end of slot t_2, p_2 must have received a certificate (i.e., $f+1$ votes) for B_2 before finalizing B_2. Therefore, there must exist a slot t_3 such that the following holds: 1) $t_1 < t_3 \leq t_2$; 2) an honest replica p_3 receives $f+1$ votes for a block B_3 of slot t_3 and B_3 is conflicting with B_1; 3) block B_3 is proposed with a valid cer_3 for the parent of B_3 and cer_3 is verified by at least one honest replica (as otherwise p_3 would receive no more than f votes for B_3). Here, $slot(\text{cer}_3) < t_3$ as cer_3 is a certificate for the parent block of B_3. W.l.o.g., suppose t_3 is the first slot such that the above holds.

Towards a contradiction, we now show that B_3 cannot be a conflicting block of B_1. Since at least $f+1$ replicas vote for B_3, there must exist an honest replica p_k among them. According to

the protocol, p_k must have a head (denoted as head_k) such that $slot(\text{head}_k) \leq slot(\text{cer}_3)$ at $\tau_{t_3} + \Delta$. Additionally, p_1 finalizes B_1 at the end of slot t_1. Therefore, according to Lemma 3, each honest replica will obtain a head from ForkChoice such that $slot(\text{head}) \geq t_1$ after $\tau_{t_1+1} + \Delta$. As $\tau_{t_3} + \Delta \geq \tau_{t_1+1} + \Delta$, we have $slot(\text{head}_k) \geq t_1$. There are two cases: $slot(\text{cer}_3) = t_1$ and $slot(\text{cer}_3) > t_1$. We begin with $slot(\text{cer}_3) = t_1$. Since p_1 finalizes B_1 at the end of slot t_1 and $slot(B_1) = t_1$, the block for cer_3 is B_1 (by Lemma 2), a contradiction. If $slot(\text{cer}_3) > t_1$, we have $t_1 < slot(\text{cer}_3) < t_3 \leq t_2$. The block for cer_3 is a conflicting block with B_1 and has been verified by at least one honest replica, a contradiction with the assumption that t_3 is the first slot such that a conflicting block is proposed.

As B_3 cannot be a conflicting block of B_1 and $slot(B_2) > slot(B_1)$, block B_2 must extend block B_1. □

Theorem 1 (Safety). *If an honest replica finalizes a block B before it finalizes another block B', then no honest replica finalizes the block B' without first finalizing B.*

Proof. Towards a contradiction, suppose an honest replica p_1 finalizes block B before it finalizes B' and another honest replica p_2 finalizes B' before it finalizes B. We assume that p_1 finalizes B at the end of a slot t_1 and p_2 finalizes B' at the end of a slot t_2. Then, there must exist a block B_1 of slot t_1 and another block B_2 of slot t_2, such that p_1 finalizes B when it finalizes B_1 and p_2 finalizes B' when it finalizes B_2. Thus, B is either B_1, an uncle block of B_1, an ancestor of B_1, or an uncle block of one of B_1's ancestors. The same relationship also holds between B' and B_2.

Since B_1 is finalized at the end of slot t_1 and B_2 is finalized at the end of slot t_2, by Corollary 1, $t_1 \neq t_2$. W.l.o.g, let $t_1 < t_2$. By Lemma 4, B_2 extends B_1. B must be an ancestor of B_2, or an uncle block of one of B_2's ancestors. As p_2 finalizes B' before it finalizes B, B' must be an ancestor of B, or an uncle block of one of B's ancestors. Therefore, p_1 must first finalize B' before it finalizes B, a contradiction.

□

Lemma 5. *If an honest leader p_1 proposes a block B_1 in an odd slot and sets the commit flag of B_1 as 1, B_1 must be voted for by all honest replicas.*

Proof. According to the protocol, p_1 must propose B_1 and include cer_c in its proposal such that $slot(\text{cer}_c) = t_1 - 1$. Since each honest replica has a head such that $slot(\text{head}) \leq t_1 - 1$ in slot t_1, each honest replica must verify and vote for B_1 when receiving B_1 and cer_c from p_1. □

Lemma 6. *If an honest leader p_1 proposes a block B_1 in an even slot, B_1 must be voted for by all honest replicas.*

Proof. According to the protocol, p_1 must have proposed a block B_i in slot $t_1 - 1$. There are two cases: p_1 receives a certificate for B_i before slot t_1; p_1 has not received a certificate for B_i.

If p_1 receives a certificate for B_i before slot t_1, p_1 must propose B_1 with a cer_c where $slot(\text{cer}_c) = t_1 - 1$ at the beginning of slot t_1. Since each honest replica has a head such that $slot(\text{head}) \leq t_1 - 1$ in slot t_1, it must verify and vote for B_1 when receiving B_1.

If p_1 receives no certificate for B_i, at least one honest replica p_k must not have voted for B_i in slot $t_1 - 1$. Since slot $t_1 - 1$ is an odd slot and p_1 is an honest leader, due to Lemma 5, the commit flag of B_i must be 0. Therefore, according to ForkChoice, the head of each honest replica remains unchanged from slot $t_1 - 1$ to slot t. Additionally, let cer_i denote the certificate for the parent of B_i. As p_k did not vote for B_i, p_k must have obtained a head_k from ForkChoice such that $slot(\text{head}_k) > slot(\text{cer}_i)$ in slot $t_1 - 1$. The other honest replicas that verified B_i must have obtained a head from ForkChoice such that $slot(\text{head}) \leq slot(\text{cer}_i)$ in slot $t_1 - 1$. Since p_k sent its cer_c to p_1 at $\tau_{t_1-1}+\Delta$ and $slot(\text{cer}_c) \geq slot(\text{head}_k)$, p_1 must obtain a cer_c from ForkChoice at the beginning of slot t_1 such that $slot(\text{cer}_c) \geq slot(\text{head})$ for all honest replicas. Then p_1 proposes B_1 with the cer_c in slot t_1. As the head of each honest replica remains unchanged from slot $t_1 - 1$ to slot t_1, each honest replica still obtains a head from ForkChoice such that $slot(\text{head}) \leq slot(\text{cer}_c)$ in slot t_1. Therefore, all honest replicas will verify and vote for B_1. □

Theorem 2 (Reorg resilience). *If an honest leader proposes a block in the synchronous network, all honest replicas will eventually finalize the block.*

Proof. Suppose that an honest leader p_1 proposes a block B_1 in a slot t_1 and sets the commit flag of B_1 as 1. According to Lemma 5 and Lemma 6, B_1 must be verified and voted for by all honest replicas. Since there are at least $f + 1$ honest replicas, at least $f + 1$ valid votes will be received by all honest replicas. As p_1 proposes no other block of slot t_1 and the commit flag of B_1 is 1, all honest replicas will finalize B_1 at the end of slot t_1.

We next prove that any block proposed by p_1 with a commit flag of 0 must be finalized by all honest replicas. According to our protocol, only blocks that p_1 proposes in odd slots could have a commit flag of 0. Suppose that p_1 proposes B_1 with a commit flag of 0 in an odd slot t_1 and proposes B_2 with a commit flag of 1 in slot $t_1 + 1$. Since the commit flag of B_2 is 1, B_2 must be finalized by all honest replicas at the end of slot t_2. Now there are two cases: p_1 receives a certificate for B_1 before slot $t_1 + 1$, or it does not. If p_1 receives a certificate for B_1 before slot $t_1 + 1$, p_1 will obtain a candidate from ForkChoice such that candidate $= B_1$ at the beginning of slot $t_1 + 1$. In this way, B_2 extends B_1 and all honest replicas will finalize B_1 when they finalize B_2. If the certificate for B_1 is not received by p_1, p_1 must obtain a cer_c from ForkChoice such that $slot(\text{cer}_c) < t_1$ at the beginning of slot $t_1 + 1$. In this way, p_1 will set the uncle block of B_2 as B_1 and all honest replicas will finalize B_1 when they finalize B_2. □

Theorem 3 (Liveness). *If an honest replica a-broadcasts a message m, then all honest replicas eventually a-deliver m.*

Proof. An honest replica can include its *a-broadcast* messages in its proposed block when it becomes a leader. According to Theorem 2, any block proposed by an honest leader will eventually be finalized by all honest replicas. Therefore, all messages *a-broadcast* by p_1 are eventually *a-delivered* by all honest replicas.

□

D Correctness of HumBird-2

Lemma 7. *If an honest replica p_i receives $2f + 1$ $\langle$commit$\rangle$ messages for a block B and another honest replica p_j receives a certificate for a block B' such that $slot(B') = slot(B)$, then $B = B'$.*

Proof. We prove the correctness by contradiction. Since p_i receives $2f+1$ ⟨commit⟩ messages for B, at least one honest replica p_1 must have sent a ⟨commit⟩ message for B. Before sending the ⟨commit⟩ message, p_1 must have received a certificate for B. In HumBird-2, a certificate contains $2f+1$ votes. Therefore, by the quorum intersection, at least one honest replica must have voted for both B' and B in $slot(B)$. However, according to our protocol, an honest replica votes for at most one block in any given slot, a contradiction. □

Corollary 2. *If an honest replica p_i receives $2f+1$ ⟨commit⟩ messages for a block B of slot t and another honest replica p_j receives $2f+1$ ⟨commit⟩ messages for a block B' of slot t, $B = B'$.*

Proof. Since p_j receives $2f+1$ ⟨commit⟩ messages for B', at least one honest replica must have sent a ⟨commit⟩ message for B'. Before sending the ⟨commit⟩ message, the replica must have received a certificate for B'. As $slot(B) = slot(B')$, by Lemma 7, $B = B'$. □

Lemma 8. *If an honest replica p_i receives $2f+1$ ⟨commit⟩ messages for a block B of slot t, at least $f+1$ honest replicas will obtain head from ForkChoice such that $slot(head) \geq t$ in any slot $t' > t$.*

Proof. If p_i receives $2f+1$ ⟨commit⟩ messages for B, at least $f+1$ honest replicas must have received a certificate for B before $\tau_t + 2\Delta$, and B must have a commit flag of 1. By the fork choice rule, these replicas must set head as B in slot $t+1$. After slot $t+1$, the replicas will not set head as any block B' where $slot(B') < slot(B)$. Therefore, these $f+1$ honest replicas will obtain a head from ForkChoice where $slot(\text{head}) \geq t$ in any slot $t' > t$. □

Lemma 9. *If an honest replica p_1 receives $2f+1$ ⟨commit⟩ messages for a block B_1 of a slot t_1 and another honest replica p_2 receives $2f+1$ ⟨commit⟩ messages for a block B_2 of a slot $t_2 > t_1$, B_2 must extend B_1.*

Proof. We prove the correctness via contradiction. As p_2 receives $2f+1$ ⟨commit⟩ messages for B_2, at least one honest replica must have sent a ⟨commit⟩ message for B_2. Before sending the message, the replica must have received a certificate for B_2. Therefore, there must exist a slot t_3 such that the following holds: 1) $t_1 < t_3 \leq t_2$; 2) an honest replica receives a certificate for block B_3 of slot t_3 and B_3 is conflicting with B_1; 3) block B_3 is proposed with a valid cer_3 for the parent of B_3 and cer_3 is verified by at least one honest replica (as otherwise no more than f ⟨vote⟩ messages for B_3 would be received). Here, $slot(\text{cer}_3) < t_3$ as cer_3 is a certificate for the parent block of B_3. W.l.o.g., suppose t_3 is the first slot such that the above holds.

Towards a contradiction, we now show that B_3 cannot be a conflicting block of B_1. Since an honest replica receives a certificate for B_3, at least $f+1$ honest replicas must have voted for B_3. According to the protocol, these replicas must have a head such that $slot(\text{head}) \leq slot(\text{cer}_3)$ in slot t_3. Additionally, p_1 receives $2f+1$ ⟨commit⟩ messages for B_1. Therefore, according to Lemma 8, at least $f+1$ honest replicas will obtain a head from ForkChoice such that $slot(\text{head}) \geq t_1$ after slot t_1. By the quorum intersection, at least one honest replica p_k must have a head in slot $t_3 > t_1$ such that $slot(\text{cer}_3) \geq slot(\text{head}) \geq t_1$. There are two cases: $slot(\text{cer}_3) = t_1$ and $slot(\text{cer}_3) > t_1$. We begin with $slot(\text{cer}_3) = t_1$. Since p_1 receives $2f+1$ ⟨commit⟩ messages for B_1, the block for cer_3 is B_1 (by Lemma 7), a contradiction. If $slot(\text{cer}_3) > t_1$, we have $t_1 < slot(\text{cer}_3) < t_3 \leq t_2$. The block for cer_3 is a conflicting block with B_1 and has been verified by at least one honest replica, a contradiction with the assumption that t_3 is the first slot such that a conflicting block is proposed.

As B_3 cannot be a conflicting block of B_1 and $slot(B_2) > slot(B_1)$, B_2 must extend B_1. □

Theorem 4 (Safety). *If an honest replica finalizes a block B before it finalizes a block B', then no honest replica finalizes the block B' without first finalizing B.*

Proof. Towards a contradiction, suppose an honest replica p_1 finalizes block B before it finalizes B' and another honest replica p_2 finalizes B' before it finalizes B. Then, there must exist a block B_1 of slot t_1 and another block B_2 of slot t_2, such that p_1 finalizes B when it receives $2f+1$ ⟨commit⟩ messages for B_1 and p_2 finalizes B' when it receives $2f+1$ ⟨commit⟩ messages for B_2. Thus, B is either B_1, an uncle block of B_1, an ancestor of B_1, or an uncle block of one of B_1's ancestors. The same relationship also holds between B' and B_2.

As p_1 receives $2f+1$ ⟨commit⟩ messages for B_1 and p_2 receives $2f+1$ ⟨commit⟩ messages for B_2, by Corollary 2, $t_1 \neq t_2$. W.l.o.g, let $t_1 < t_2$. By Lemma 9, B_2 extends B_1. B must be an ancestor of B_2, or an uncle block of one of B_2's ancestors. As p_2 finalizes B' before it finalizes B, B' must be an ancestor of B, or an uncle block of one of B's ancestors. Therefore, p_1 must first finalize B' before it finalizes B, a contradiction. □

Lemma 10. *After GST, if an honest leader proposes a block B_1 in an odd slot and sets the commit flag of B_1 as 1, B_1 must be voted for by all honest replicas.*

The proof is similar to that in Lemma 5.

Lemma 11. *After GST, if an honest leader proposes a block B_1 in an even slot, B_1 must be voted for by all honest replicas.*

The proof is similar to that in Lemma 6.

Theorem 5 (Reorg resilience). *If an honest leader proposes a block after GST, all honest replicas will eventually finalize the block.*

Proof. Suppose that an honest leader p_1 proposes a block B_1 in a slot t_1 and sets the commit flag of B_1 as 1. According to Lemma 10 and Lemma 11, B_1 must be verified and voted for by all honest replicas. Since there are at least $2f+1$ honest replicas, each honest replica will observe a valid certificate for B_1 before $\tau_{t_1} + 2\Delta$. As the commit flag of B_1 is 1, each honest replica will send a ⟨commit⟩ message for block B_1. Therefore, all honest replicas will receive $2f+1$ ⟨commit⟩ messages for block B_1 and finalize B_1.

We next prove that any block proposed by p_1 with a commit flag of 0 must be finalized by all honest replicas. According to our protocol, only blocks that p_1 proposes in odd slots could have a commit flag of 0. Suppose that p_1 proposes B_1 with a commit flag of 0 in an odd slot t_1 and proposes B_2 with a commit flag of 1 in slot t_1+1. Since the commit flag of B_2 is 1, all honest replicas will receive $2f+1$ ⟨commit⟩ messages for B_2 and finalize B_2. Now there

are two cases: p_1 receives a certificate for B_1 before slot $t_1 + 1$, or it does not. If p_1 receives a certificate for B_1 before slot $t_1 + 1$, p_1 will obtain a candidate from ForkChoice such that candidate $= B_1$ at the beginning of slot $t_1 + 1$. In this way, B_2 extends B_1 and all honest replicas will finalize B_1 when they finalize B_2. If the certificate for B_1 is not received by p_1, p_1 must obtain a cer_c from ForkChoice such that $slot(\text{cer}_c) < t_1$ at the beginning of slot $t_1 + 1$. In this way, p_1 will set the uncle block of B_2 as B_1 and all honest replicas will finalize B_1 when they finalize B_2. □

Theorem 6 (Liveness). *If an honest replica proposes a message m, then all honest replicas eventually finalize m.*

Proof. An honest replica can include its proposed messages in its proposed block when it becomes a leader after GST. According to Theorem 5, any block proposed by an honest leader after GST will eventually be finalized by all honest replicas. Therefore, all messages proposed by p_1 are eventually finalized by all honest replicas. □

E Additional Evaluation Figures

We show the latency vs. throughput of HumBird-1, SRR-ST, and RL-SHS under the failure scenario where the faulty replicas are selected randomly in Figure 11 and Figure 12. The results are consistent with our observations in Sec. 6: HumBird-1 consistently achieves higher peak throughput than all other protocols and degrades more gracefully as the number of faulty replicas increases.

F Analysis of the Expected Latency

In this section, we provide formal analysis of the expected latency for HumBird-1, HumBird-2, Sync Streamlet, and Partially Sync Streamlet. We assume each protocol operates under a random leader rotation model, where every leader is randomly selected from all n replicas with equal probability.

Theorem 7. *HumBird-1 has an expected latency of* 7.5Δ.

Proof. Suppose a transaction tx is submitted to all honest replicas right before the beginning of an odd slot t. We denote the leader of slot $t + 2i$ as L_i where $i \in \mathbb{N}$. According to HumBird-1, L_i is also the leader of slot $t + 2i + 1$. We use B_i to denote the block proposed by L_i in slot $t + 2i$. For each i, L_i must be an honest replica with independent probability $p = \frac{n-f}{n} \geq \frac{1}{2}$. Let X be the number s.t. L_X is the first honest leader since slot t. Here, X is a random variable and its expected value is $E[X] = \frac{1}{p} - 1$. By definition, if $X > 0$, L_i is faulty for all $0 \leq i < X$. Since L_X is honest, B_X will contain tx if tx has not been included in the ancestors of B_X.

According to our protocol, all honest replicas enter slot $t + 2X$ by time $\tau_t + X \cdot 4\Delta$. When $X > 0$, every honest replica will finalize B_X by time $\tau_t + X \cdot 4\Delta + 4\Delta$. When $X = 0$, the finalization time of B_0 depends on whether the leader preceding L_0 is honest. In particular, all honest replicas will finalize B_0 by time $\tau_t + 2\Delta$ with probability p (if the preceding leader is honest), and by time $\tau_t + 4\Delta$ with probability $1 - p$ (if the preceding leader is faulty). Therefore, the expected latency of HumBird-1 is:

$$
\begin{aligned}
E &= P(X=0)\cdot[p(2\Delta)+(1-p)(4\Delta)] + \sum_{k=1}^{\infty} P(X=k)\cdot(4\Delta+4k\Delta) \\
&= p(2p\Delta+4\Delta-4p\Delta) + 4\Delta\underbrace{\sum_{k=1}^{\infty}P(X=k)}_{1-p} + 4\Delta\underbrace{\sum_{k=1}^{\infty}kP(X=k)}_{E[X]} \\
&= \frac{4\Delta}{p} - 2p^2\Delta
\end{aligned} \tag{1}
$$

As $p \geq \frac{1}{2}$, $E \leq 7.5\Delta$. □

Theorem 8. *HumBird-2 has an expected latency of* $\frac{28}{9}\Delta + 3\delta$.

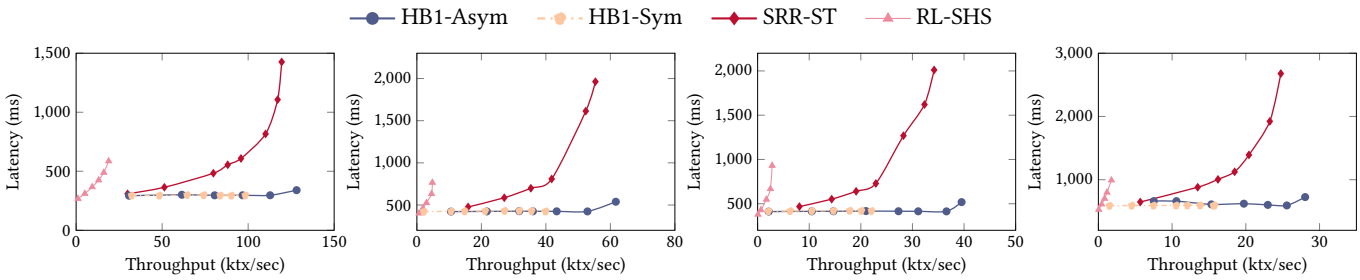
Proof. Suppose a transaction tx is submitted to all honest replicas right before the beginning of an odd slot t. We denote the leader of slot $t + 2i$ as L_i where $i \in \mathbb{N}$. According to HumBird-2, L_i is also the leader of slot $t + 2i + 1$. We use B_i to denote the block proposed by L_i in slot $t + 2i$. For each i, L_i must be an honest replica with independent probability $p = \frac{n-f}{n} \geq \frac{2}{3}$. Let X be the number s.t. L_X is the first honest leader since slot t. Here, X is a random variable and its expected value is $E[X] = \frac{1}{p} - 1$. By definition, if $X > 0$, L_i is faulty for all $0 \leq i < X$. Since L_X is honest, B_X will contain tx if tx has not been included in the ancestors of B_X.

According to our protocol, all honest replicas enter slot $t + 2X$ by time $\tau_t + X \cdot 4\Delta$. When $X > 0$, every honest replica will finalize B_X by time $\tau_t + X \cdot 4\Delta + 2\Delta + 3\delta$. When $X = 0$, the finalization time of B_0 depends on whether the leader preceding L_0 is honest. In particular, all honest replicas will finalize B_0 by time $\tau_t + 3\delta$ with probability p (if the preceding leader is honest), and by time $\tau_t + 2\Delta + 3\delta$ with probability $1 - p$ (if the preceding leader is faulty). Therefore, the expected latency of HumBird-2 is:

$$
\begin{aligned}
E &= P(X=0)\cdot[p(3\delta)+(1-p)(2\Delta+3\delta)] \\
&\quad + \sum_{k=1}^{\infty}P(X=k)\cdot(4k\Delta+2\Delta+3\delta) \\
&= p[3p\delta+(1-p)(2\Delta+3\delta)] + (2\Delta+3\delta)\underbrace{\sum_{k=1}^{\infty}P(X=k)}_{1-p} \\
&\quad + 4\Delta\underbrace{\sum_{k=1}^{\infty}kP(X=k)}_{E[X]} \\
&= 3\delta - 2\Delta(1+p^2) + \frac{4\Delta}{p}
\end{aligned} \tag{2}
$$

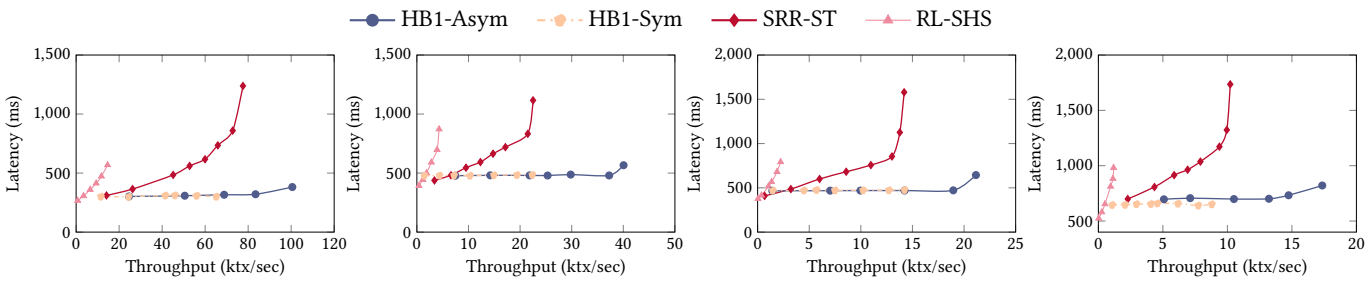
As $p \geq \frac{2}{3}$, $E \leq \frac{28}{9}\Delta + 3\delta$. □

Now, we analyze the expected latency of Sync Streamlet and Partially Sync Streamlet. We begin with an overview of the two protocols. In particular, both protocols proceed in synchronized epochs. Every epoch lasts for 2Δ time and is assigned to a random leader. At the beginning of each epoch, the leader of the epoch proposes a block extending the longest notarized chain. Replicas vote



(a) $n = 5$. **(b)** $n = 21$. **(c)** $n = 41$. **(d)** $n = 61$.

Figure 11: Latency vs. throughput of HumBird-1, SRR-ST, and RL-SHS under the failure scenario where the number of faulty replicas is $\lfloor \frac{n}{4} \rfloor$ and faulty replicas are selected randomly.



(a) $n = 5$. **(b)** $n = 21$. **(c)** $n = 41$. **(d)** $n = 61$.

Figure 12: Latency vs. throughput of HumBird-1, SRR-ST, and RL-SHS under the failure scenario where the number of faulty replicas is $\lfloor \frac{n-1}{2} \rfloor$ and faulty replicas are selected randomly.

for the block if it extends their locally observed longest notarized chains. A block is notarized once it collects votes from a quorum of replicas.

In Sync Streamlet, if a replica observes a notarized chain where the last six blocks have consecutive epoch numbers and no conflicting notarized blocks exist at the same heights as the six blocks, the first one in the last six blocks is finalized. According to Theorem 9 in [21], when there are eight consecutive epochs $e, e+1, ..., e+7$ all with honest leaders, replicas can observe a notarized chain where the last six blocks have consecutive epoch numbers $e+2, ..., e+7$ by the end of epoch $e+7$. Meanwhile, no conflicting notarized blocks exist at the same heights as the six blocks. In this case, the block of epoch $e+2$ will be finalized.

In Partially Sync Streamlet, if a replica observes a notarized chain where the last three blocks have consecutive epoch numbers, the second one in the three blocks is finalized. According to Theorem 6 in [21], when there are five consecutive epochs $e, e+1, ..., e+4$ all with honest leaders, replicas can observe a notarized chain where the last three blocks have consecutive epoch numbers $e+2, ..., e+4$ by the end of epoch $e+4$. In this case, the block of epoch $e+3$ will be finalized.

Theorem 9. *Sync Streamlet has an expected latency of* 1016Δ.

Proof. Suppose an epoch e begins at time t and a transaction tx is submitted to all honest replicas by time t. We use L_i to denote the leader of epoch $e+i$, where $i \in \mathbb{N}$, and B_i to denote the block proposed by L_i. For each i, L_i must be an honest replica with independent probability $p = \frac{n-f}{n} \geq \frac{1}{2}$. We use E_n to denote the expected number of epochs required to reach the first sequence of n consecutive honest leaders since epoch e. By modeling the leader election as a sequence of independent Bernoulli trials, we can derive that $E_n = \frac{1-p^n}{(1-p)p^n}$.

At the beginning of epoch e, there are three possible cases:

- **Case** HH**:** The leaders of epoch $e-1$ and $e-2$ were honest (probability p^2).
- **Case** MH**:** The leader of epoch $e-2$ was faulty but the leader of epoch $e-1$ was honest (probability $(1-p)p$).
- **Case** M**:** The leader of epoch $e-1$ was faulty (probability $1-p$).

In Case HH, if all leaders from epoch e to $e+5$ are honest, tx will be included in B_0 and finalized by the end of epoch $e+5$. Otherwise, tx will be finalized after the first sequence of eight consecutive honest leaders occurs. In particular, the finalization time of tx is the end of the epoch for the eighth leader in this sequence. Therefore, the expected latency of tx in Case HH is:

$$\begin{aligned} E_{HH} &= 6p^6 \cdot 2\Delta + (1-p)(1+E_8) \cdot 2\Delta + p(1-p)(2+E_8) \cdot 2\Delta \\ &\quad + p^2(1-p)(3+E_8) \cdot 2\Delta + p^3(1-p)(4+E_8) \cdot 2\Delta \\ &\quad + p^4(1-p)(5+E_8) \cdot 2\Delta + p^5(1-p)(6+E_8) \cdot 2\Delta \\ &= 2\Delta \cdot [(1-p^6)E_8 + \frac{1-p^6}{1-p}] \end{aligned} \tag{3}$$

In Case MH, if all leaders from epoch e to $e+6$ are honest, tx will be finalized by the end of epoch $e+6$. Otherwise, replicas

will finalize tx after encountering eight consecutive honest leaders. Therefore, the expected latency of tx in Case MH is:

$$E_{MH} = 2\Delta \cdot [(1-p^7)E_8 + \frac{1-p^7}{1-p}] \tag{4}$$

In Case M, since the leader of epoch $e-1$ was faulty, replicas will finalize tx after encountering eight consecutive honest leaders. Therefore, the expected latency of tx in Case M is:

$$E_M = 2\Delta \cdot E_8 \tag{5}$$

Combining these results, the total expected latency is:

$$E = p^2 \cdot E_{HH} + (1-p)p \cdot E_{MH} + (1-p) \cdot E_M \tag{6}$$

As $p \geq \frac{1}{2}$ and $E_8 = \frac{1-p^8}{(1-p)p^8}$, $E \leq 1016\Delta$. □

Theorem 10. *Partially Sync Streamlet has an expected latency of* $\frac{537\Delta}{16}$.

Proof. Suppose an epoch e begins at time t and a transaction tx is submitted to all honest replicas by time t. We use L_i to denote the leader of epoch $e+i$, where $i \in \mathbb{N}$, and B_i to denote the block proposed by L_i. For each i, L_i must be an honest replica with independent probability $p = \frac{n-f}{n} \geq \frac{1}{2}$. We use E_n to denote the expected number of epochs required to reach the first sequence of n consecutive honest leaders since epoch e. By modeling the leader election as a sequence of independent Bernoulli trials, we can derive that $E_n = \frac{1-p^n}{(1-p)p^n}$.

At the beginning of epoch e, there are four possible cases:

- **Case** HHH**:** The leaders of epochs $e-1$, $e-2$, and $e-3$ were honest (probability p^3).
- **Case** MHH**:** The leader of epoch $e-3$ was faulty but the leaders of epoch $e-1$ and $e-2$ were honest (probability $(1-p)p^2$).
- **Case** MH**:** The leader of epoch $e-2$ was faulty but the leader of epoch $e-1$ was honest (probability $(1-p)p$).
- **Case** M**:** The leader of epoch $e-1$ was faulty (probability $1-p$).

In Case HHH, if all leaders from epoch e to e+1 are honest, tx will be included in B_0 and finalized by the end of epoch $e+1$. Otherwise, tx will be finalized after the first sequence of five consecutive honest leaders occurs. In particular, the finalization time of tx is the end of the epoch for the fifth leader in this sequence. Therefore, the expected latency of tx in Case HH is:

$$\begin{aligned} E_{HHH} &= 2p^2 \cdot 2\Delta + (1-p)(1+E_5) \cdot 2\Delta + p(1-p)(2+E_5) \cdot 2\Delta \\ &= 2\Delta \cdot [(1-p^2)E_5 + \frac{1-p^2}{1-p}] \end{aligned} \tag{7}$$

In Case MHH, if all leaders from epoch e to $e+2$ are honest, tx will be finalized by the end of epoch $e+2$. Otherwise, replicas will finalize tx after encountering five consecutive honest leaders. Therefore, the expected latency of tx in Case MHH is:

$$E_{MHH} = 2\Delta \cdot [(1-p^3)E_5 + \frac{1-p^3}{1-p}] \tag{8}$$

In Case MH, if all leaders from epoch e to $e+3$ are honest, tx will be finalized by the end of epoch $e+3$. Otherwise, replicas will finalize tx after encountering five consecutive honest leaders. Therefore, the expected latency of tx in Case MH is:

$$E_{MH} = 2\Delta \cdot [(1-p^4)E_5 + \frac{1-p^4}{1-p}] \tag{9}$$

In Case M, since the leader of epoch $e-1$ was faulty, replicas will finalize tx after encountering five consecutive honest leaders. Therefore, the expected latency of tx in Case M is:

$$E_M = 2\Delta \cdot E_5 \tag{10}$$

Combining these results, the total expected latency is:

$$E = p^3 \cdot E_{HHH} + (1-p)p^2 \cdot E_{MHH} + (1-p)p \cdot E_{MH} + (1-p) \cdot E_M \tag{11}$$

As $p \geq \frac{2}{3}$ and $E_5 = \frac{1-p^5}{(1-p)p^5}$, $E \leq \frac{537\Delta}{16}$. □